НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

спеціальності 121 Інженерія програмного забезпечення

на тему: «Асинхронна черга завдань для бібліотеки AsyncIO Python»

Виконала:
студентка IV курсу, групи КП-62
Рябоконь Тетяна Олексіївна _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Заболотня Тетяна Миколаївна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри ММСА ІПСА, к.т.н., доцент,
Дідковська Марина Віталіївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____2019 р.

ЗАВДАННЯ

на дипломний проєкт студентці

Рябоконь Тетяні Олексіївні

1. Тема проєкту «Асинхронна черга завдань для бібліотеки AsyncIO Python», керівник проєкту Заболотня Тетяна Миколаївна, к.т.н., доцент, затверджені наказом по університету від «22» травня 2020 р. № 1181-с.

2. Термін подання студенткою проєкту «15» червня 2020 р.

3. Вихідні дані до проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки:
   - аналіз версії мови програмування Python та технологій розроблення менеджерів завдань;
   - розроблення асинхронної черги завдань для бібліотеки AsyncIO Python;
   - опис використаних шаблонів проєктування та протоколів передачі повідомлень;
   - аналіз розробленої черги завдань.

5. Перелік обов'язкового графічного матеріалу:
   - алгоритм роботи виконавця та планувальника завдань (креслення);
   - компоненти бібліотеки (креслення);
   - схема взаємодії бібліотеки Tabasco та RabbitMQ (плакат);
   - схема роботи RabbitMQ (плакат).

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання «31» жовтня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	14.11.2019	
2.	Розроблення та узгодження технічного завдання	28.11.2019	
3.	Розроблення архітектури бібліотеки асинхронної черги завдань	15.12.2019	
4.	Підготовка матеріалів першого розділу дипломного проєкту	30.12.2019	
5.	Розроблення структури окремих частин бібліотеки та взаємодії між ними	03.02.2020	
6.	Підготовка матеріалів другого розділу дипломного проєкту	20.02.2020	
7.	Програмна реалізація асинхронної черги завдань	10.03.2020	
8.	Тестування асинхронної черги завдань	17.03.2020	
9.	Підготовка матеріалів третього розділу дипломного проєкту	30.03.2020	
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	11.04.2020	
11.	Підготовка графічної частини дипломного проєкту	21.04.2020	
12.	Оформлення документації дипломного проєкту	26.05.2020	

Студент Тетяна РЯБОКОНЬ

Керівник проєкту Тетяна ЗАБОЛОТНЯ

АНОТАЦІЯ

Даний дипломний проект присвячений розробленню програмної бібліотеки асинхронної черги завдань для бібліотеки asyncio Python.

Дана бібліотека являє собою менеджер завдань, що розроблений для використання при розробленні додатків на асинхронному Python, та призначена для виконання завдань як в реальному часі, так і для планування періодичних завдань.

У роботі виконано аналіз існуючих на даний момент програмних рішень, розроблених відомими компаніями, для вирішення даної проблеми. Розроблена бібліотека підтримує можливості сучасного Python 3, використовує asyncio та нові ключові слова async/await у Python 3.6+. Також вона покрита анотаціями типів та перевірена за допомогою mypy.

Бібліотека є надійною і легко переживає проблеми з мережею та збої на сервері. У разі втрати з’єднання з брокером стан черги буде автоматично відновлений. Також дана бібліотека використовує для своєї роботи систему обміну повідомленнями між компонентами програмної системи – брокер повідомлень RabbitMQ.

У даному дипломному проекті розроблено: архітектуру бібліотеки асинхронної черги завдань, програмну реалізацію планувальника та виконавця завдань, та інтерфейс командного рядка.

ABSTRACT

This project is dedicated to the development of a software library of asynchronous task queues for the asyncio Python library.

This library is a task manager designed for use in developing applications with asynchronous Python and is designed to perform tasks both in real time and for scheduling periodic tasks.

The analysis of the existing software libraries for solving this problem is performed in the work. The developed library supports the capabilities of modern Python 3, uses asyncio and new keywords async/await in Python 3.6+. It is also covered with type annotations and checked with mypy.

The library is reliable and easily experiences network problems and server failures. If the connection with the broker is lost, the queue status will be restored automatically. Also, this library uses for its work a system of messaging between components of the software system - message broker RabbitMQ.

In this project was developed: the architecture of the asynchronous task queue library, the software implementation of the task scheduler and worker, and the command line interface.

АННОТАЦИЯ

Данный дипломный проект посвящен разработке программной библиотеки асинхронной очереди задач для библиотеки asyncio Python.

Данная библиотека представляет собой менеджер задач, который разработан для использования при разработке приложений на асинхронном Python, и предназначена для выполнения задач как в реальном времени, так и для планирования выполнения периодических задач.

В работе выполнен анализ существующих на данный момент программных решений, разработанных известными компаниями, для решения данной проблемы. Разработанная библиотека поддерживает возможности современного Python 3, использует asyncio и новые ключевые слова async/await в Python 3.6+. Также она покрыта аннотациями типов и проверена с помощью mypy.

Библиотека является надежной и легко переживает проблемы с сетью и сбои на сервере. В случае потери соединения с брокером, состояние очереди будет автоматически восстановлено. Также данная библиотека использует для своей работы систему обмена сообщениями между компонентами программной системы - брокер сообщений RabbitMQ.

В данном дипломном проекте разработаны: архитектура библиотеки асинхронной очереди задач, программная реализация планировщика и исполнителя задач, а также интерфейс командной строки.

ДП.045440-01-90 Програмне забезпечення для зберігання та використання персональних документів. Відомість проєкту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проєкту		
ДП.045440-02-91	Асинхронна черга завдань	5	
	для бібліотеки AsyncIO		
	Python. Технічне завдання		
ДП.045440-03-81	Асинхронна черга завдань	63	
	для бібліотеки AsyncIO		
	Python. Пояснювальна		
	записка		
ДП.045440-04-51	Асинхронна черга завдань	4	
	для бібліотеки AsyncIO		
	Python. Програма та		
	методика тестування		
ДП.045440-05-33	Асинхронна черга завдань	12	
	для бібліотеки AsyncIO		
	Python. Керівництво		
	програміста		
ДП.045440-06-99	Асинхронна черга завдань	1	
	для бібліотеки AsyncIO		
	Python. Алгоритм роботи		
	виконавця та		
	планувальника завдань.		
	UML-діаграма діяльності		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045440-07-99	Асинхронна черга завдань	1	
	для бібліотеки AsyncIO		
	Python. Компоненти		
	бібліотеки. UML-діаграма		
	компонентів		
ДП.045440-08-98	Асинхронна черга завдань	1	
	для бібліотеки AsyncIO		
	Python. Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2019 р.

АСИНХРОННА ЧЕРГА ЗАВДАНЬ ДЛЯ БІБЛІОТЕКИ ASYNCIO PYTHON

Технічне завдання

ДП.045440-02-91

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Тетяна РЯБОКОНЬ

2019

ЗМІСТ

1. Найменування та галузь застосування…….3
2. Підстава для розроблення…….....3
3. Призначення розробки……...3
4. Вимоги до програмного продукту…….…3
5. Вимоги до проєктної документації…….…..4
6. Етапи проєктування……....5
7. Порядок тестування розробки……...5

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Асинхронна черга завдань для бібліотеки AsyncIO Python.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проєктування, затверджене кафедрою програмного забезпечення комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для використання при розробленні програмних додатків на асинхронному Python, а саме на базі модулю стандартної бібліотеки AsyncIO. Так як асинхронний Python ще молодий, є потреба в написанні бібліотек, які будуть працювати в асинхронному стилі. Бібліотека забезпечує виконання завдань, які потрібно виконати в фоновому режимі та розвантажити сервер з основним процесом додатку, або для виконання періодичних завдань.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Бібліотека повинна забезпечувати такі основні функції:

1) можливість виконання завдань у реальному часі
2) можливість налаштування виконання періодичних завдань;
3) масштабування виконавців завдань для збільшення відмовостійкості;
4) можливість розширення структури бібліотеки;
5) робота з брокером повідомлень;
6) повинна працювати з AsyncIO Python.

Розробку виконати на Python 3.5+ з використання модуля AsyncIO.

Додаткові вимоги:

1) має бути надійною і легко переживати проблеми з мережею та збої на сервері;
2) має бути покрита анотаціями типів та перевірена за допомогою mypy;
3) має легко розгортатися на будь-якій операційній системі;
4) повинна мати інтерфейс керування у вигляді CLI – утиліти командного рядку;

5. ВИМОГИ ДО ПРОЄКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проєкту повинна бути розроблена наступна документація:

1) пояснювальна записка;
2) програма та методика тестування;
3) керівництво програміста;
4) креслення:
   - «Алгоритм роботи виконавця та планувальника завдань. UML-діаграма діяльності»;
   - «Компоненти бібліотеки. UML-діаграма компонентів».

6. ЕТАПИ ПРОЄКТУВАННЯ

Вивчення літератури за тематикою роботи……..14.11.2019

Розроблення та узгодження технічного завдання……....28.11.2019

Розроблення архітектури бібліотеки……..……...15.12.2019

Розроблення структури окремих частин бібліотеки та взаємодії між ними ……..……..……..……..……..……..03.02.2020

Програмна реалізація асинхронної черги завдань……...17.03.2020

Тестування бібліотеки…….....03.04.2020

Підготовка матеріалів текстової частини проєкту……..….28.04.2020

Підготовка матеріалів графічної частини проєкту……..…12.05.2020

Оформлення технічної документації проєкту……..…25.05.2020

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

АСИНХРОННА ЧЕРГА ЗАВДАНЬ ДЛЯ БІБЛІОТЕКИ ASYNCIO PYTHON

Пояснювальна записка

ДП.045440-03-81

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____Тетяна РЯБОКОНЬ

2020

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 3
ВСТУП 5
1. ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ 7
1.1. Огляд проблеми, яка вирішується ПЗ 7
1.2. Аналіз вимог до розроблюваного ПЗ 8
1.3. Аналіз існуючих рішень 8
1.4. Результати проведеного аналізу 13
2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ 15
2.1. Вибір мови програмування для розроблення 15
2.2. Вибір черги повідомлень, на базі якої буде працювати бібліотека 16
3. СТРУКТУРНО-АЛГОРИТМІЧНА РЕАЛІЗАЦІЯ 22
3.1. Аналіз вимог до програмних засобів 22
3.2. Логічна структура розроблюваної системи 33
4. ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ 42
4.1. Опис використаних інструментів, мови та сторонніх залежностей 42
4.2. Аспекти реалізації структурно-алгоритмічної організації бібліотеки 45
4.3. Приклади використання бібліотеки 53
4.4. Опис використаного інструментального програмного забезпечення 58
ВИСНОВКИ 60
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ 61
ДОДАТКИ 63

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Асинхронне програмування – підхід до оброблення вводу/виводу, що дозволяє програмі працювати з мережевими пристроями та обслуговувати запити мережі, або працювати з базами даних, поки процесор продовжує виконувати програму, не очікуючи на відповідь від мережі або бази даних.

Фреймворк – це програмне забезпечення, що орієнтоване на розроблення програмних додатків та потрібне для їх швидкого створення та розгортання.

CLI – вид програмного інтерфейсу взаємодії між людиною та комп'ютером, де команди передаються комп'ютерній програмі у вигляді рядків тексту.

Python – високорівнева інтерпретована мова програмування загального призначення, орієнтована на збільшення продуктивності програміста та легкої читаємості коду.

Python Enhancement Proposal (PEP) – це документи з пропозиціями щодо розвитку мови Python, їхне створення є основним механізмом для пропозиції нових можливостей і для документування проєктних рішень, що були запроваджені в Python.

async/await – синтаксична особливість багатьох мов програмування, в тому числі Python, що дозволяє створювати асинхронні неблокуючі функції так, як звичайні синхронні функції.

Asyncio – це бібліотека для написання конкурентного коду за допомогою синтаксису async/await, є частиною стандартної бібліотеки Python.

Муру – інструмент для статичної перевірки типів для Python, який має на меті поєднати переваги динамічної та статичної типізації.

Flake8 – інструмент, що дозволяє перевіряти код і виявляти в ньому стилістичні помилки і порушення різних конвенцій коду на Python.

Python Package Index (PyPi) – є офіційним сховищем стороннього програмного забезпечення для Python. Деякі менеджери пакетів, включаючи pip, використовують PyPI як джерело за замовчуванням для пакетів та їх залежностей.

Брокер повідомлень – додаток, який перетворює та передає повідомлення по одному протоколу від програми-джерела, в повідомлення протоколу додатка-приймача, тим самим виступаючи між ними посередником.

RabbitMQ – це програмне забезпечення брокер повідомлень з відкритим кодом, яке реалізує протоколи Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (STOMP) , MQ Telemetry Transport (MQTT) та інші.

Apache Kafka – це програмна платформа з відкритим вихідним кодом, що написана на Scala та Java.

Redis – резидентна база даних класу NoSQL, що працює з структурами даних типу ключ-значення.

Celery – асинхронна черга завдань із відкритим кодом. Хоча вона підтримує планування завдань, основна увага приділяється операціям у режимі реального часу.

Faust – це Python бібліотека, яка забезпечує як обробку потоків, так і обробку подій подібно до таких інструментів, як Kafka Streams або Apache Spark.

RQ (Redis Queue) – це проста бібліотека Python для встановлення завдань в чергу на опрацювання та обробки їх у фоновому режимі з працівниками.

Docker – це програмне забезпечення, що має модель платформа як послуга, і використовується для управління ізольованими Linux-контейнерами.

ВСТУП

Центральна частина комп’ютеру, яка виконує команди, з яких складається кожна програма, називається центральним процесором. Зазвичай програми будуються так, що процесор зайнятий, доки програма не завершить свою роботу. Швидкість, з якою програма виконає певні обчислювальні операції, цілком залежить від швидкості процесора.

Але сучасні програми також взаємодіють з багатьма речами, які не залежать від процесора комп’ютера, на якому вони виконуються. Наприклад, програми можуть спілкуватися по комп'ютерній мережі або запитувати дані з жорсткого диску, що набагато повільніше, ніж отримувати їх з оперативної пам’яті. Не ефективно давати процесору простоювати, коли виконуються такі операції, в цей час ним може бути виконана корисна робота. Частково цю проблему вирішує операційна система – вона перемикає процесор між декількома запущеними програмами, але це не вирішує проблему, коли ми хочемо, щоб певна програма продовжувала виконуватись, поки вона чекає на запит мережі.

В синхронному програмуванні всі команди виконуються одна за одною. Результат функції, яка виконує тривалу дію, буде повернений лише тоді, коли дія закінчиться і функція зможе обчислити результат. Це зупиняє програму на час виконання цієї дії. Натомість асинхронна модель дозволяє робити кілька операцій одночасно. Коли починається тривала дія, програма продовжує виконувати інші операції, а, коли дія завершується, програма інформується про це і отримує доступ до результату. Таким чином, процесор не простоює та виконує корисну дію в той час як виконуються операції вводу-виводу.

Разом із цим були прийняті декілька пропозицій щодо покращення мови програмування Python, в яких йдеться про новий модуль стандартної бібліотеки asyncio. Тепер на рівні мови в Python підтримується асинхронний підхід до написання коду.

Так як асинхронний Python ще молодий, є потреба в написанні бібліотек, які будуть працювати в асинхронному стилі, тому що зараз більшість великих бібліотек є синхронними, а більшість з тих, що підтримують неблокуючу поведінку, ще молоді та експериментальні. Наприклад, зараз немає великої гнучкої та стабільної бібліотеки асинхронної черги завдань. Складно уявити собі великі веб-додатки, які не виконують тривалі операції, що не потребують участі користувача. Зазвичай виконання таких операцій автоматизується в певне завдання, яке відправляється в чергу завдань. Такий програмний інструментарій є дуже важливим і повинен бути стабільним, добре масштабованим та легко інтегруватися з основним додатком.

Випливаючи з цього, метою даної роботи є створення асинхронної черги завдань для асинхронного Python, що буде працювати на базі модулю стандартної бібліотеки asyncio.

1. ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ

1.1. Огляд проблеми, яка вирішується ПЗ

Розробникам веб-додатків часто доводиться стикатися з типовими завданнями, такими як, надсилання електронного листа користувачеві або обробка завантажених даних. Найчастіше такі задачі не вимагають участі кінцевого користувача проєкту, тобто їх можна виконувати у фоновому режимі. Якщо реалізовувати такі задачі в одному з процесів веб-сервера, його робота буде сповільнена, збільшиться час відгуку і погіршиться UX.

Як результат були створені спеціальні бібліотеки, які узагальнено можна називати – розподілена черга завдань. Дані рішення мають свої переваги, такі як: інтеграції з великими Python фреймворками, системи моніторингу, велика спільнота розробників та ін.

В PEP 3156 Asynchronous IO Support Rebooted: the “asyncio” Module в стандартній бібліотеці мови Python з’являється модуль asyncio, тобто підхід до розробки змінюється, адже модуль asyncio дозволяє втілювати підхід асинхронного програмування [1]. Відповідно, існуючі рішення не підходили для асинхронного Python. З часом з’явилися бібліотеки, які працюють з asyncio, та їх ще дуже мало і вони мають певні особливості, які можуть ускладнювати їх використання [2].

Аналізуючи існуючі бібліотеки, можна дійти висновку, що для асинхронного Python ще немає такого рішення, яке б не використовувало складні розподілені брокери повідомлень, не потребувало розгортання додаткових баз даних, використовувало для роботи саме чергу і було б простим у використанні.

Випливаючи з цього, метою даної роботи є створення бібліотеки-асинхронної черги завдань для модулю asyncio Python.

1.2. Аналіз вимог до розроблюваного ПЗ

На основі визначеної мети проєкту були сформовані вимоги, яким має відповідати асинхронна черга завдань для модулю asyncio Python.

1. Черга має бути оформлена як бібліотека та імпортуватися у будь-яку існуючу програму Python.
2. Бібліотека має підтримувати можливості сучасного Python 3, повністю використовувати переваги asyncio та нові ключові слова async/await у Python 3.6+ для запуску декількох потокових процесорів в одному процесі, поряд з веб-серверами та іншими мережевими службами. Також вона має бути покрита анотаціями типів та перевірена за допомогою mypy.
3. Бібліотека має бути надійною і легко переживати проблеми з мережею та збої на сервері. У разі виходу з ладу вузла, він має автоматично відновитись, також повинні бути резервні вузли, які приймуть роботу на себе одразу ж. Має бути передбачена можливість масштабування.
4. Бібліотека має бути легко розширюваною, абстрагувати сховища, серіалізатори та навіть транспортування повідомлень, щоб розробникам було зручно розширити її новими можливостями та інтегруватись до існуючих систем.
5. Має використовувати для своєї роботи систему обміну повідомленнями між компонентами програмної системи – брокер повідомлень.

Згідно із запропонованими вимогами проведено детальний порівняльний аналіз існуючих програмних рішень. Також враховано їх переваги та недоліки при розробленні програмного застосунку.

1.3. Аналіз існуючих рішень

При розгляді питання про створення асинхронної черги завдань було встановлено, що найбільш популярним рішенням для передачі

повідомлень є використання RabbitMQ. Ця технологія потрібна для організації взаємодії між процесами та координації в програмних системах [3]. Інші системи можуть використовувати Redis або Apache Kafka.

Також було проаналізовано бібліотеки черги завдань, розроблені відомими компаніями.

1.3.1. Аналіз програмного рішення – Faust

Дане програмне рішення є бібліотекою оброблення потоків і переносить ідеї з Kafka Streams в Python. Ця система не використовує DSL і це означає, що можна використовувати популярні бібліотеки Python при обробці потоку: NumPy, PyTorch, Pandas, NLTK, Django, Flask, SQLAlchemy та ін. Бібліотеці Faust потрібен Python 3.6 або пізнішої версії для нового синтаксису async/await та підтримки анотацій типів. Це робить Faust єдиним прямим аналогом, через те що система підтримує можливості асинхронного програмування в Python [4].

Його концепція «агентів» походить від моделі «актора», а це означає, що потоковий процесор може працювати одночасно на багатьох ядрах процесора та на сотнях машин одночасно.

Faust використовує Kafka брокер та RocksDB, що робить його складним інструментом, бо Kafka є більш складною та потужною ніж RabbitMQ, тому підготувати Faust для роботи складніше, ніж бібліотеки, які використовують інші системи, наприклад, RabbitMQ або Redis [5].

Faust тісно пов'язаний з основними концепціями Кафки. Система спирається на те, як Kafka керує групами споживачів, щоб визначити, чи не використовують вони певну тему повідомлень, і запустити агент у випадку, коли програма повідомляє про її функціонування. Тобто Kafka призначає цей розділ агенту, який Kafka досі ідентифікує як активний у групі споживачів.

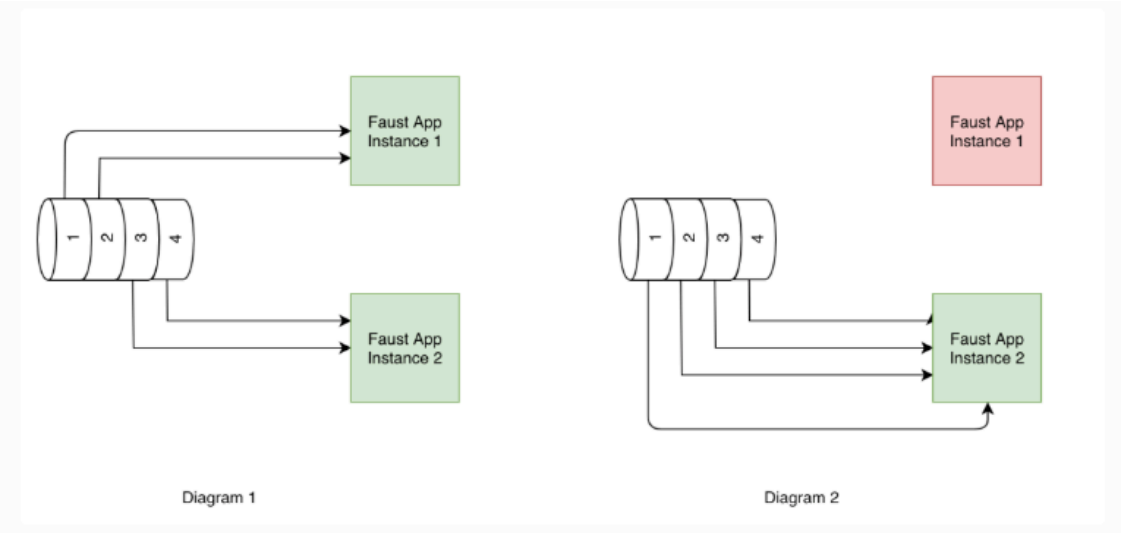


Рис. 1.1. Ілюстрація призначення Faust розділу агенту, що живе в додатку, який Kafka ідентифікує як активний у групі споживачів

На зображенні вище ми маємо єдину тему повідомлень в Kafka, яка складається з чотирьох розділів. У нас також є два екземпляри одного і того ж додатка Faust, що працюють одночасно. На Діаграмі 1 обидва екземпляри працюють, тому обраний для цієї теми лідер Faust вирішить, які теми отримає він, решту тем отримають інші споживачі в тій же групі. В даному випадку він присвоює 2 розділи кожному екземпляру.

На діаграмі 2 перший екземпляр вимикається. Через деякий час група клієнтів повідомить лідера про те, що один із споживачів не працює, і змусить перерозподілити теми. В цьому випадку він призначить усі чотири розділи одному і тому ж екземпляру.

Із врахуванням визначених раніше вимог для даного програмного рішення виділено наступне: перевагами системи є підтримка модулю asyncio Python, надійність та розширюваність. Недоліками є те, що підготувати Faust для роботи складніше, ніж, наприклад, бібліотеки на базі черги на основі стандарту AMQP, що може бути критично, при розробленні маленьких додатків.

1.3.2. Аналіз програмного рішення Celery

Celery є асинхронною чергою завдань на основі розподіленої системи обміну повідомлень. Вона орієнтована на роботу в режимі реального часу, але також підтримує планування [6].

Завдання виконуються одночасно на одному або декількох робочих серверах, використовуючи багатопроцесорність, Eventlet або gevent. Завдання можуть виконуватися асинхронно або синхронно.

Зараз Celery не підтримує модуль asyncio Python, в офіційній документації зазначено, що наступна основна версія Celery підтримуватиме версію Python 3.5 та вище, для забезпечення підтримки asyncio. Відмова від підтримки Python 2 дозволить розробникам бібліотеки видалити велику кількість коду сумісності, а перехід на Python 3.5 дозволить використовувати анотації типів, ключові слова async/await, модуль стандартної бібліотеки asyncio та інші поняття, яких немає в старих версіях Python.

Celery працює на базі брокера повідомлень, який використовується для комунікації між клієнтами та виконавцями завдань. Для ініціювання завдання клієнт додає повідомлення до черги, тоді брокер доставляє це повідомлення виконавцю.

Система Celery може складатися з декількох виконавців і брокерів для забезпечення високої доступності та горизонтального масштабування. Celery написана на Python, але протокол може бути реалізований будь-якою мовою. Окрім Python, існує клієнт для Node.js та клієнт для PHP.

Celery потребує системи обміну повідомлень, наприклад, RabbitMQ або Redis. Також існує підтримка безлічі інших експериментальних рішень, включаючи використання, наприклад, SQLite для локальної розробки.

Celery може працювати на одній машині, на декількох машинах або навіть між різними дата-центрами.

Дане програмне рішення проаналізовано відповідно до описаних вище вимог. Щодо переваг, то система використовує як систему обміну повідомленнями RabbitMQ або Redis, що є зручним для розробників, адже налаштування роботи цих систем є простішим, ніж налаштування Kafka. Головним недоліком є те, що Celery не підтримує модуль asyncio.

1.3.3. Аналіз програмного рішення Dramatiq

Dramatiq – це бібліотека для обробки завдань для Python з акцентом на простоту, надійність та продуктивність [7].

Основною причиною створення Dramatiq, є прагнення зробити бібліотеку розподіленої черги завдань простою.

Характеристики Dramatiq такі:

- висока надійність та продуктивність;
- просте і легке для розуміння ядро.

Розробники позиціонують бібліотеку, як простіший та зрозуміліший конкурент Celery. В офіційній документації зазначено, що на відміну від Celery, Dramatiq має просту реалізацію, має можливість надавати завданням пріоритет (навідміну від Celery та RQ) та має покращений принцип роботи з відкладеними завданнями.

Як систему обміну повідомлень Dramatiq використовує також RabbitMQ або Redis. Dramatiq не підтримує модуль asyncio і є багатопотоковим.

З точки зору обраних вимог для аналізу програмного рішення, дана система має ряд переваг, а саме: просту реалізацію, використовує RabbitMQ, як брокер повідомлень та має механізм пріоритизації завдань. Бібліотека не підтримує модуль asyncio та є не дуже популярною серед розробників, тому не розвивається активно.

1.3.4. Аналіз програмного рішення RQ

RQ (Redis Queue) – це бібліотека Python для встановлення завдань на виконання в чергу та обробки їх додатками-виконавцями у фоновому режимі [8]. Бібліотека використовує Redis. Натхненням для створення цієї бібліотеки стали Celery та Resque для Ruby. Основна ідея бібліотеки – бути легкою у використанні та інтегруванні до основного веб-додатку альтернативою великим бібліотекам розподіленим чергам завдань [8].

Через просту реалізацію бібліотека залишилася без багатьох корисних функцій:

- пріоритизація завдань;
- відкладене виконання завдань;
- завдання розкладу виконання завдань;
- автоматичні повторення виконання завдань та багато іншого.

Але головним недоліком є те, що RQ не підтримує модуль asyncio.

1.4. Результати проведеного аналізу

Виходячи з отриманих даних після проведення аналізу, можна стверджувати, що згадані системи повністю не вирішують поставлену задачу. Більшість із описаних рішень не підтримують модуль asyncio Python, а те рішення, яке все ж працює з асинхронним Python, є достатньо складним у використанні та підтримці. З іншого боку є рішення, які занадто прості і не вирішують всі задачі, які поставлені перед бібліотекою для обробки розподіленої черги завдань.

Отже, для програмного рішення розподіленої черги завдань з підтримкою модулю asyncio виділено наступні вимоги:

- підтримка асинхронного Python;
- висока надійність та продуктивність;
- проста та зрозуміла реалізація;
- проста для розуміння та легка для налаштування система обміну повідомленнями;

- повнота функцій бібліотеки для обробки розподіленої черги завдань (пріорітизація завдань, відкладене виконання завдань, завдання розкладу виконання завдань, автоматичні повторення виконання завдань та ін).

Таблиця 1.1

Результати аналізу існуючих програмних рішень

	Faust	Dramatiq	Celery	RQ
Працює з asyncio Python	+	–	–	–
Покриття анотаціями типів	+	–	–	–
Проста реалізація	–	+	–	+
Автоматичне повторення виконання завдання	+	+	–	–
Відкладене виконання завдань	+	+	+	–
Завдання розкладу виконання завдань	+	–	+	–
Підтримка RabbitMQ	–	+	+	–
Підтримка багатьох брокерів повідомлень	–	+	+	–

2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Вибір мови програмування для розроблення

Згідно з поставленою задачею, асинхронна черга завдань розробляється для асинхронного Python, а саме для модулю стандартної бібліотеки asyncio. Модуль asyncio є частиною стандартної бібліотеки тільки з версії Python 3.4, тому для розробки потрібна найновіша стабільна версія мови. В Python 3.8 з’являється:

- підтримка asyncio в інтерактивній оболонці і тепер await можна використовувати безпосередньо з самої оболонки;
- доданий моржовий оператор “:=”, який присвоює значення змінній, як частина більшого виразу;
- asyncio.run() переміщено до стабільного API, зараз ця функція може використовуватися для виконання підпрограми та повернення результату при автоматичному керуванні циклом подій;
- задачі asyncio тепер можна називати, наприклад, передавши іменований аргумент “name” в asyncio.create_task(). Ім'я завдання видно у виводі repr() asyncio.Task і його також можна отримати методом get_name (), що дуже допомагає при відладці коду;
- додано підтримку Happy Eyeballs для asyncio.loop.create_connection(), Happy Eyeballs - це алгоритм, який може підвищити швидкість відповіді для веб-додатків, що розуміють як IPv4, так і IPv6, для користувачів, які намагаються встановити з'єднання одночасно через обидві версії інтернет протоколу;
- додано оператор “=” до f-строк для документування певних виразів та для більш зручного відлагодження програми;

- проведено багато важливих оптимізацій, завдяки яким робота з деякими модулями або типами даних стала в рази швидшою та потребує менше пам'яті.

На даний момент версія Python 3.8 є найновішою стабільною версією мови та має деякі важливі зміни, що стосуються саме asyncio, тому для розробки нової бібліотеки бажано використовувати саме її [9].

2.2. Вибір черги повідомлень, на базі якої буде працювати бібліотека

Відповідно до описаних вимог, асинхронна черга завдань для асинхронного Python повинна використовувати для своєї роботи систему обміну повідомленнями. По своїй суті, система обміну повідомленнями або брокер повідомлень – це програма яка транслює повідомлення з формального протоколу відправника до формального протоколу повідомлень отримувача. Основна мета брокера – це приймати вхідні повідомлення з програм та виконувати деякі дії над ними, наприклад, він може використовуватися для:

- управління чергою навантаження або чергою повідомлень для декількох приймачів, забезпечуючи надійне зберігання, гарантовану доставку повідомлень і, можливо, управління транзакціями;
- направлення повідомлення до одного або декількох напрямків;
- виконування агрегацію повідомлень, розкладати повідомлення на кілька повідомлень та надсилати їх до місця призначення, після чого компонувати відповіді в одне повідомлення для повернення користувачеві;
- взаємодії із зовнішнім сховищем, щоб доповнити повідомлення або зберегти його;
- реагування на події чи помилки;

- маршрутизації повідомлень на основі теми за шаблоном публікації-підписки.

Існує дуже багато різних брокерів повідомлень на основі різних протоколів, деякі з них потрібно розгортати власноруч, деякі знаходяться в хмарі. Для розроблення черги завдань виберемо один з найбільш стабільних, достатньо масштабованих інструментів з великою спільнотою розробників. Розглянемо такі інструменти: RabbitMQ, Apache Kafka, Redis.

2.2.1. Система обміну повідомленнями Apache Kafka

Apache Kafka – розподілений програмний брокер повідомлень, проєкт з відкритим вихідним кодом, що розробляється в рамках фонду Apache. Написаний на мові програмування Scala і Java [10].

Для цієї системи варто відзначити такі переваги:

- потужна платформа обробки потоків подій;
- відмовостійкість і надійність;
- хороша масштабованість;
- безкоштовний продукт з великою спільнотою розробників;
- мультіарендність – елемент архітектури програмного забезпечення, де єдиний екземпляр додатку, що запущений на сервері, обслуговує багато організацій-клієнтів;
- підходить для роботи в режимі реального часу;
- відмінно підходить для проєктів з великими даними.

Недоліками даної системи є:

- відсутність готових до використання рішень – розширень, бібліотек, тощо;
- відсутність повного набору для моніторингу;
- відсутність маршрутизації;
- проблеми зі збільшенням кількості повідомлень.

За допомогою Apache Kafka можна успішно створювати керовані даними додатки та керувати складними бекенд-системами. Apache Kafka вміє:

- публікувати та підписуватися на потоки записів з чудовою масштабованістю та продуктивністю, що робить його придатним для використання у великих проєктах;
- надійно зберігати потоки, поширюючи дані по декількох вузлах для високо доступного розгортання;
- обробляти потоки даних по мірі їх надходження, що дозволяє агрегувати, виконувати з'єднання даних у потоці тощо.

Отже, поєднуючи всі ці функції, можна зробити висновок, що Kafka не є звичайним брокером повідомлень. Це потужна платформа обробки потоків подій, що здатна обробляти трильйони повідомлень на день, але вимагає багато зусиль з налаштування її роботи через брак готових рішень.

2.2.2. Система обміну повідомленнями RabbitMQ

RabbitMQ – це програмний брокер повідомлень з відкритим кодом, який спочатку реалізовував протокол розширеної черги повідомлень AMQP, але з тих пір його архітектуру було розширено підтримкою інших протоколів обміну повідомленнями.

Для цієї системи варто відзначити такі переваги, як:

- підходить для багатьох мов програмування та протоколів обміну повідомленнями;
- може використовуватися в різних операційних системах та хмарних середовищах;
- простий для початку використання та розгортання;
- дає можливість використовувати різні інструменти для розробників;
- має сучасний вбудований інтерфейс користувача;
- пропонує кластеризацію і дуже добре працює з нею;

- масштабується до 500 000+ повідомлень в секунду.

Окрім переваг зазначимо такі недоліки:

- не транзакційний за замовчуванням;
- конфігурація, яку можна зробити за допомогою коду, є мінімальною;
- має проблеми з обробленням великих обсягів даних.

RabbitMQ створений як брокер повідомлень загального використання і базується на шаблоні комунікації публікація-підписка, процес обміну повідомленнями може бути синхронним або асинхронним. Отже, основними особливостями брокера повідомлень є:

- підтримка численних протоколів, змінна маршрутизація до черг, різні типи обміну;
- розгортання кластерів забезпечує високу доступність та пропускну здатність, також програмне забезпечення може використовуватися в різних зонах та регіонах;
- можливості використання Puppet та Docker для розгортання та сумісність з найпопулярнішими сучасними мовами програмування;
- підключена автентифікація, підтримка TLS та LDAP.

Основна перевага цього брокера повідомлень – ідеальний набір розширень у поєднанні з досить великою масштабованістю, детальна документація, а також можливість роботи з різними моделями обміну повідомленнями, наприклад, такими як:

- модель прямого обміну;
- модель обміну темами (кожен споживач отримує повідомлення, яке надсилається на певну тему);
- модель обміну Fanout (повідомлення отримують усі споживачі, підключені до черги).

2.2.3. Система обміну повідомленнями на основі Redis

Redis – це розподілене сховище пар ключ-значення із відкритим вихідним кодом, яке зберігається в оперативній пам'яті і може функціонувати як брокер повідомлень, база даних та кеш-пам'ять. Redis підтримує різні види абстрактних структур даних, такі як рядки, списки, карти, множини, впорядковані множини та бітові карти. Redis швидкий і легкий, що робить його популярним серед розробників [11].

Як переваги, для цієї системи можна виділити:

- Redis є швидким та масштабованим, оскільки працює з оперативною пам’яттю;
- надзвичайно простий у налаштуванні, використанні та розгортанні;
- є резидентною СУБД та надає розширений кеш ключ-значення;
- має велику кількість бібліотек-клієнтів на різних мовах.

Але також є і певні недоліки:

- Redis був створений з іншими намірами, а не для того, щоб бути брокером повідомлень, він підтримує основні операції необхідні для брокера повідомлень, однак для потужної маршрутизації повідомлень Redis не є найкращим варіантом;
- характеризується високою затримкою в роботі з великими повідомленнями, тому Redis краще підходить для невеликих повідомлень.

Для реалізації системи обміну повідомлення використовують шаблон публікації-підписки – Redis Pub/Sub.

Після огляду найпопулярніших систем обміну повідомленнями, можна сказати, що кожна з них певною мірою підходить для розроблення на її основі асинхронної черги завдань, адже кожна має певні переваги та недоліки. RabbitMQ підтримує кластеризацію та може забезпечити більшу масштабованість та доступність, ніж Redis. RabbitMQ надає більше можливостей для користувацьких налаштувань: права доступу, тривалість

певного обміну або черги, гарантія доставки повідомлення, наприклад, підтримка транзакцій. Якщо ж порівнювати Kafka та RabbitMQ, то обидві системи можуть приймати кілька мільйонів повідомлень в секунду, хоча їх архітектура відрізняється, і кожна працює в певних умовах краще. RabbitMQ контролює свої повідомлення майже в оперативній пам'яті, використовуючи великий кластер (30+ вузлів). Натомість Kafka використовує операції послідовного вводу-виводу, і, таким чином, вимагає менше апаратного забезпечення. Kafka поступається RabbitMQ можливостю використовувати багато плагінів та готових рішень, які економлять час розробника та прискорюють розробку. Завдяки ним можна легко налаштувати фільтри, пріоритети, упорядкувати повідомлення тощо. Як і Kafka, RabbitMQ вимагає розгортання та управління програмним забезпеченням, але він має зручний вбудований інтерфейс і дозволяє використовувати SSL для більш безпечного з'єднання. Що стосується відмовостійкості, то тут RabbitMQ поступається Kafka.

Отже, після проведеного аналізу можна зробити висновок, що RabbitMQ більше за інші рішення підходить для розроблення асинхронної черги завдань, адже ця система має достатньо велику спільноту розробників, багато готових рішень та плагінів, вбудований інтерфейс користувача, добре справляється з навантаженнями та може забезпечити гарантію доставки повідомлення і є відносно простою для розгортання.

3. СТРУКТУРНО-АЛГОРИТМІЧНА РЕАЛІЗАЦІЯ

3.1. Аналіз вимог до програмних засобів

Беручи до уваги поставлене завдання на реалізацію асинхронної черги завдань для модуля asyncio Python, були обґрунтовані вимоги до розроблення цієї програмної системи. Для того щоб чітко розуміти, чого саме треба досягти та яку функціональність система повинна мати, треба дослідити проблеми, які постають перед розроблюваним програмним додатком, визначити цілі та результати, що повинні бути досягнуті після реалізації цієї системи.

3.1.1. Визначення проблем та цілей, що постають перед асинхронною чергою завдань

Головною проблемою, яку повинна вирішувати черга завдань, є великий час виконання деяких запитів до програмних систем та їх можливе перевантаження. Наприклад, після оформлення замовлення в Інтернет-магазині, користувач повинен отримати лист з підтвердженням замовлення, якщо ця дія буде виконуватись синхронно під час запиту на оформлення, користувач буде довго очікувати на відповідь системи, що є поганим UX. Також системі не потрібно знати результат виконання операції про відправлення листа, щоб оформити замовлення, тому роботу щодо відправлення листа можна довірити черзі завдань, яка виконає цю дію асинхронно і значно збільшить швидкість відповіді. Щодо можливого перевантаження програмних систем, воно може виникати через такі проблеми як:

- завдання може бути виконано тільки, якщо частина системи, яку воно використовує, доступна, в іншому випадку завдання не може бути виконано і буде втрачене, що означає можливу втрату цінних даних кінцевих користувачів;

- неможливість якісно контролювати стан виконання завдань, які суттєво навантажують систему, що ускладнює моніторинг системи та налагодження коду при розробці;
- реалізація відкладеного виконання завдань або виконання за розкладом є складною, що є дуже не зручним для високонавантажених систем;
- складна реалізація міжпроцесної взаємодії.

Підсумовуючи вищезазначене, можна скласти дерево проблем, яке наведене на рис. 3.1.

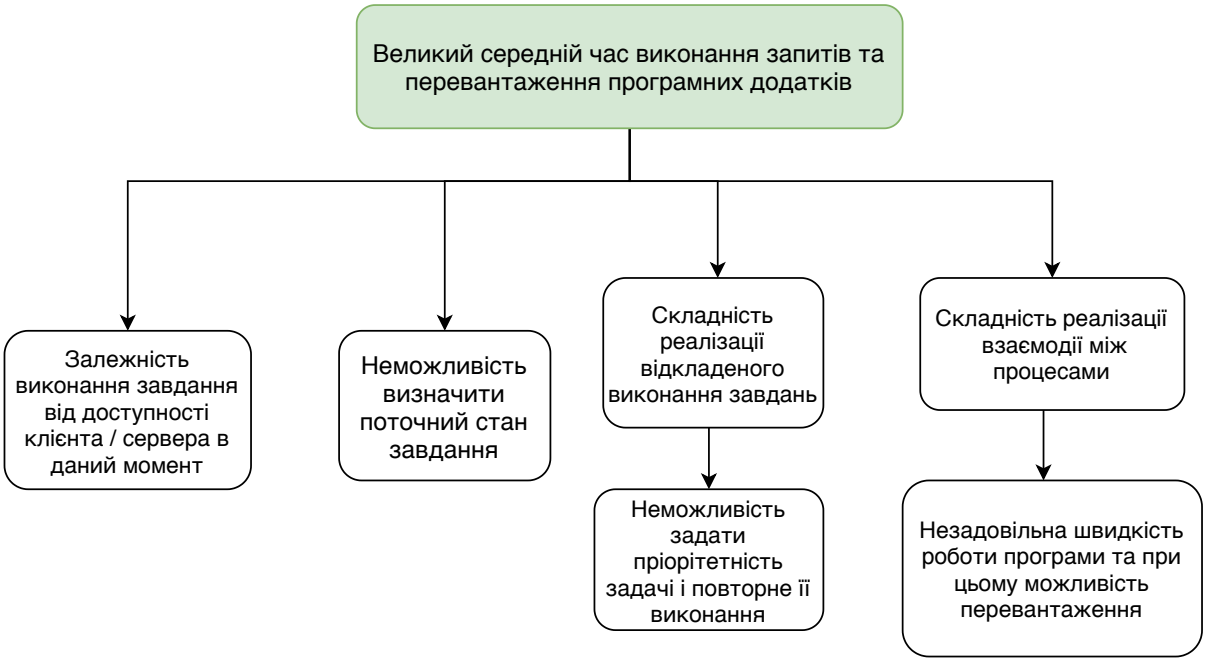


Рис. 3.1. Візуалізація дерева проблем, які повинна вирішувати асинхронна черга завдань

За результатами дослідження проблематики було сформовано цілі, що мають бути досягнуті як результат розроблення даної програмної системи. Метою цього проєкту є розроблення саме для асинхронного Python черги завдань, яка буде відповідати всім стандартам та зможе стати гідним аналогом відомих синхронних бібліотек.

Визначені цілі представлені на рис. 3.2.

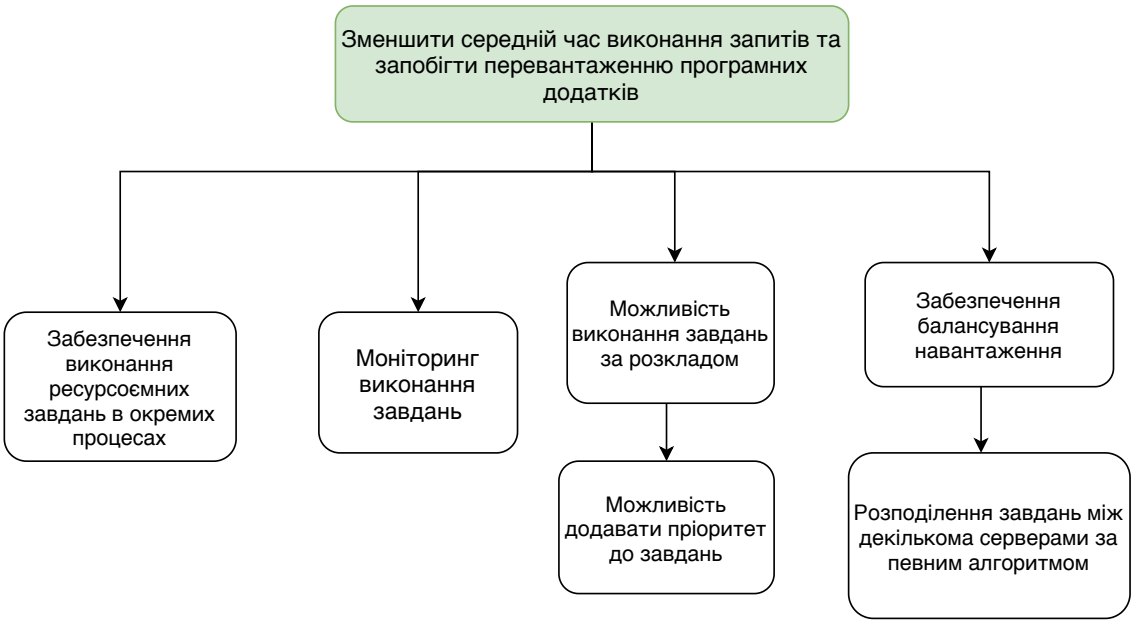


Рис. 3.2. Візуалізація дерева цілей, які повинні бути досягнуті в ході розроблення асинхронної черги завдань

Взявши до уваги визначені цілі, було сформовано дерево результатів, які повинні бути досягнуті в процесі розроблення даної програмної системи.

Визначені результати представлені на рис 3.3.

Рис. 3.3. Візуалізація дерева результатів програмної системи

3.1.2. Обґрунтування вимог до асинхронної черги завдань

Вимоги до програмного забезпечення – це сфера програмної інженерії, яка займається встановленням потреб зацікавлених сторін, що повинні вирішуватися розроблюваним програмним забезпеченням. В словнику IEEE Standard Glossary of Software Engineering Terminology надається таке визначення терміну “вимоги до програмного забезпечення”:

- умова або здатність, що необхідна користувачеві для вирішення проблеми або досягнення мети;
- умова або здатність, яку повинна надавати система або її компонент, щоб задовольнити контракт, стандарт, специфікацію чи інший офіційно накладений документ;
- задокументоване зображення умов або можливостей, як для пункту 1, так і для пункту 2 [12].

Діяльність, пов’язана з роботою над вимогами до програмного забезпечення, може бути розбита на збір вимог, їх аналіз, специфікацію, перевірку правильності вимог та управління.

Розглянемо кожен етап докладніше:

1. Збір вимог – це дослідження та виявлення того, що потрібно від розроблюваної системи користувачам, клієнтам та іншим зацікавленим сторонам.
2. Аналіз вимог зосереджується на визначенні потреб та умов для створення нового або зміненого проєкту, враховуючи вимоги різних зацікавлених сторін. Проводиться аналіз, документація, перевірка та управління вимогами до програмного забезпечення. На цьому етапі вимоги деталізуються до рівня, достатнього для проєктування системи.
3. Специфікація – це аналіз програмної системи, яку потрібно розробити, що моделюється після специфікації вимог зацікавлених сторін. Специфікація вимог до програмного забезпечення встановлює функціональні та нефункціональні

вимоги, і може також включати набір випадків використання, які програмне забезпечення повинно вміти коректно обробляти.

4. Перевірка включає методи підтвердження того, що правильний набір вимог був визначений для створення рішення, яке задовольняє цілі проєкту.
5. Управління вимогами є безперервним процесом протягом усього проєкту і включає в себе документування, аналіз, відстеження, визначення пріоритетів та узгодження вимог, а після – контроль змін та комунікацію з відповідними зацікавленими сторонами.

Для визначення конкретних вимог до програмного забезпечення необхідно спочатку визначити зацікавлені сторони проєкту. Зацікавленими сторонами проєкту є всі, хто має значний інтерес до його розроблення. До зацікавлених сторін відносяться:

- будь-хто, хто управляє системою (звичайні користувачі та оператори обслуговування);
- кожен, хто отримує вигоду з системи (функціональні, політичні, фінансові та соціальні бенефіціари);
- будь-хто, хто інвестує в систему;
- організації, що регулюють різні аспекти системи (фінансові, аспекти безпеки та інші);
- люди чи організації, які є конкурентами розробленої системи
- організації, відповідальні за системи, які взаємодіють із розробленою системою;
- організації, які горизонтально інтегруються з організацією, для якої проєктується система.

Для програмної бібліотеки асинхронної черги завдань для asyncio Python зацікавленими сторонами є розробники проєктів, які пишуться на Python 3 та використовують асинхронний підхід до програмування.

Після проведення аналізу вимог до програмного забезпечення, визначено ряд вимог до розроблюваної системи. Вимоги були сформовані в результаті дослідження аналогічних систем та опитування користувачів, які підходять в якості зацікавлених сторін. Визначені вимоги були коротко відображені в таблиці, яка наведена нижче. В наступних розділах деякі з них будуть розглянуті більш детально.

Таблиця 3.1

Вимоги до програмного забезпечення

Код вимоги	Назва вимоги	Аналіз вимоги
F1	Вимоги до оформлення	Код повинен бути написаний зрозуміло та бути легко підтримуваним. Програма повинна бути добре структурована, а імена змінних, методів та класів інтуїтивно зрозумілими
F2	Вимоги до реалізації	Бібліотека повинна бути написана на Python3 з використанням модулю стандартної бібліотеки asynico та покрита анотаціями типів.
F3	Вимоги до сторонніх залежностей	Бібліотека повинна мати тільки обов’язкові для її роботи залежності, щоби зменшити шанс на появу вразливостей.
F4	Вимоги до інтерфейсу завдання	Зрозуміле визначення завдання, наприклад, Python-декоратор та зручна передача аргументів завданню.
F5	Вимоги до ізольованості даних	Підтримка каналів з RabbitMQ. Канали потрібні для забезпечення можливості розділення підключення до брокера на декілька логічних підключень.

Продовження табл. 3.1

F6	Вимоги до надійності зв’язку	Стан з'єднання буде відновлено після перепідключення. Після повторного встановлення з'єднання канали, черги та обміни повідомленнями будуть відновлені.
F7	Вимоги до відмовостійкості	Розподілення навантаження, завдяки реалізації шаблону Master/Worker, що дозволяє розподілити завдання між декількома виконавцями.
F8	Вимоги до періодичних завдань	Бібліотека має дозволяти виконувати завдання періодично та мати зручний інтерфейс для конфігурації часу його виконання.
F9	Вимоги до інтерфейсу	Система повинна мати інтерфейс керування у вигляді CLI-утиліти командного рядку.
F10	Вимоги до виконання завдань	Оскільки повідомлення надсилаються клієнтам асинхронно, зазвичай в каналі в будь-який момент часу є більше одного повідомлення, тому повинна бути можливість обмежувати кількість повідомлень, які можуть бути оброблені одночасно, щоб уникнути проблеми обмеженого буфера (англ. bounded-buffer problem) на стороні споживача.
F11	Вимоги до крос-платформності	Бібліотека повинна працювати на різних операційних системах.
F12	Вимоги до поширення	Бібліотека має бути опублікована у каталозі програмного забезпечення PyPI для зручного встановлення і використання

Продовження табл. 3.1

F13	Вимоги до підтримки	Бібліотека повинна бути опублікована на сервісі хостингу та спільної розробки програмного забезпечення – GitHub.

3.1.3. Вимоги до інтерфейсу завдання

Завдання – це базова складова додатку асинхронної черги завдань, об'єкт, який можна створити з будь-якого об'єкта в Python, що підтримує спеціальний метод __call__, тобто може бути викликаний подібно до функції. Об’єкт завдання в розроблюваній системи повинен визначати що відбувається в той момент, коли з клієнтського коду було “поставлене” завдання (повідомлення надіслане до брокеру повідомлень), та за те, що повинно відбутись, коли виконавець отримає це повідомлення.

Кожен об’єкт завдання повинен мати унікальне ім'я, яке посилається у повідомленнях, щоб виконавець міг знайти потрібну функцію для виконання.

Повідомлення, яке містить завдання не повинно видалятися з черги, доки виконавець не взяте виконавцем до роботи.

Завдання повинні створюватися легко, наприклад, за допомогою Python-декоратору. Також кожне завдання повинно мати унікальну назву, тому декоратор завдань створить ім'я, що базуватиметься на модулі, в якому визначено завдання, та назві функції завдання.

Повинна бути передбачена можливість зареєструвати створене завдання в додатку асинхронної черги завдань.

Аргументи завданню повинні передаватись, як і аргументи для звичайної Python функції.

3.1.4. Вимоги до ізольованості даних

Деякі програми потребують декількох логічних підключень до брокера повідомлень. Однак тримати багато з’єднань TCP одночасно відкритими небажано, оскільки це споживає системні ресурси та ускладнює налаштування брандмауерів. Підключення протоколу AMQP 0-9-1 мультиплексуються каналами, які можна розглядати як "легкі з'єднання, які розділяють одне з'єднання TCP".

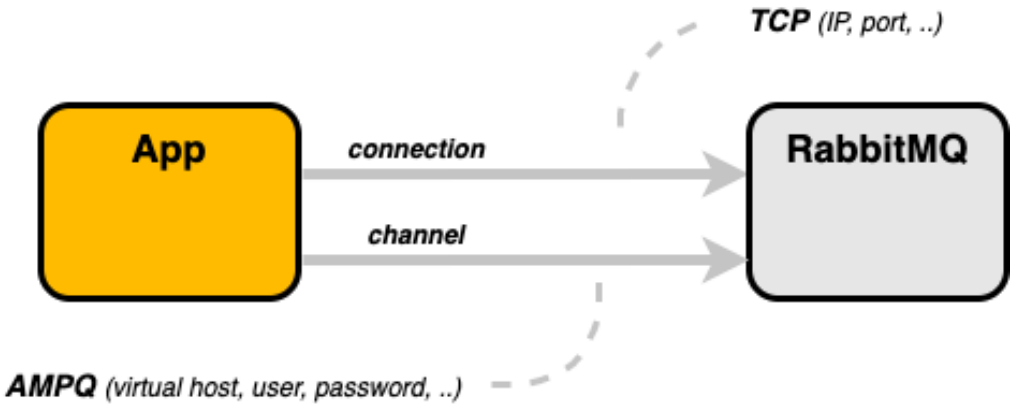


Рис. 3.3. Різниця між AMPQ з’єднанням та каналом

Кожна операція протоколу, яку виконує клієнт, відбувається на певному каналі. Зв'язок на певному каналі повністю відокремлений від зв'язку на іншому каналі, тому кожен метод протоколу також містить ідентифікатор каналу, ціле число, яке використовують як брокер, так і клієнти, щоб визначити, для якого каналу звертається метод. Канал існує лише в контексті зв'язку і ніколи не є самостійним.

Оскільки RabbitMQ реалізує описану функціональність каналів, розроблювана асинхронна черга завдань повинна мати відповідну обгортку для роботи за каналами. Це може бути корисно якщо додаток потребує логічного розділення завдань на певні теми [3].

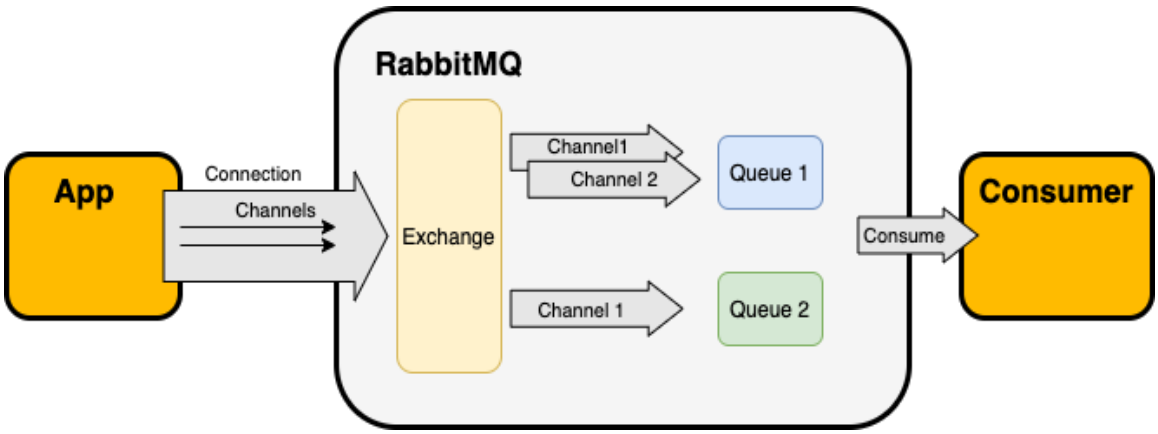


Рис. 3.4 Схема роботи RabbitMQ з каналами

3.1.5. Вимоги до надійності зв'язку

Проблеми з мережевим підключенням та перевантаженість, мабуть, є найпоширенішим класом відмов для систем обміну повідомленнями. Наприклад, мережа може вийти з ладу, або брандмауери можуть перервати з'єднання, які вони вважають непрацюючими. Всі проблеми з мережею потребують часу на виявлення та виправлення.

Окрім збоїв у підключенні, сервер та клієнтські програми можуть бути недоступними через некоректну роботу обладнання або помилку в програмному забезпеченні.

Цей перелік збоїв, звичайно, зовсім не вичерпний. Він не охоплює більш тонкі помилки, такі як погіршення продуктивності, зловмисні програми чи програми з помилками, що виснажують систему ресурсів тощо. Такі збої можна виявити за допомогою моніторингу та перевірок стану здоров'я.

Коли з'єднання не вдається, повідомлення можуть знаходитися в режимі транзиту між клієнтом і сервером, такі повідомлення не доставлятимуться – їх потрібно буде повторно передавати. Спеціальний сигнал серверу – підтвердження – повідомляє сервер та клієнти коли це зробити. Підтвердження використовуються в обох напрямках – для повідомлення сервера про доставку та обробку повідомлення.

TCP гарантує, що пакети були доставлені до отримувача, і будуть повторно передаватись до тих пір, поки збої на мережевому рівні не будуть оброблені.

Зважаючи на це, розроблювана бібліотека має надавати функціональність, яка автоматично відновлює підключення після збоїв. Це також означає, що стан з'єднання буде відновлено після підключення: канали, черги та обміни з їх прив'язкою будуть відновлені.

3.1.6. Вимоги до відмовостійкості

Поширена проблема паралельного програмування полягає в тому, як збалансувати обчислювальне навантаження між різними завданнями для одночасного виконання. Для паралельних програм, що не мають зв'язку між завданнями (або нечастою, але добре структурованою комунікацією) та ефективним рішенням з «автоматичним балансуванням динамічного навантаження» визначається єдиний майстер, який керує збором завдань та результатів. Потім набір виконувачів захоплює завдання, виконує роботу та відправляє результати назад майстру, а потім бере наступне завдання. Це триває, поки не будуть обчислені всі завдання. Такий підхід називається шаблоном Master/Worker і часто використовується для багатопотокових програм, в яких потрібно вирішити декілька примірників однієї проблеми.

Шаблон Master/Worker можна використовувати, щоб забезпечити певний рівень відмовостійкості. Майстер підтримує дві черги: одну для завдань, які ще потрібно призначити, і другу для завдань, які вже були призначені виконавцю, але не виконані. Коли перша черга буде порожньою, майстер може додатково призначати завдання з черги незавершених завдань. Тому, якщо виконавець помирає і не може виконати свої завдання, інший виконавець зможе взяти в роботу його незавершені завдання.

Таким чином, розроблювана черга завдань повинна реалізовувати шаблон Master/Worker для забезпечення відмовостійкості та перерозподілення навантаження [13].

3.1.7. Вимоги до виконання завдань

Протокол AMQP 0-9-1 визначає метод basic.qos, щоб зробити можливим обмежувати кількість непідтверджених повідомлень на каналі (або з'єднанні) при обробці повідомлення. Обмеження можна налаштувати за допомогою значення “prefetch count”. Як тільки число досягне налаштованої кількості, RabbitMQ припинить надсилати нові повідомлення на канал, доки хоча б одне з них не буде підтверджено [3].

Отже, розроблювана бібліотека повинна реалізовувати можливість налаштування параметру “prefetch count”.

3.2. Логічна структура розроблюваної системи

Провівши дослідження проблем, які повинна вирішувати розроблювана система, визначивши цілі та результати, яких треба досягти в ході розробки та проаналізувавши вимоги до продукту, було створено відповідну логічну структуру розробки.

Розроблювана бібліотека має складатися з таких модулів:

- beat – планувальник періодичних завдань;
- config – містить структури даних, що описують конфігурації модулів;
- crontab – модуль, що відповідає за встановлення періодичного завдання та конфігурацію його виконання;
- task – модуль, що представляє саме завдання та його конфігурації;
- worker – модуль, який обробляє завдання;
- app – головний модуль, що відповідає за встановлення конфігурацій для всього додатку, встановлює з'єднання з брокером та надає інтерфейс для керування задачами;

- cli – модуль керування додатком за допомогою командної строки.

Нижче наведена візуалізація взаємодії між цими модулями (рис. 3.5).

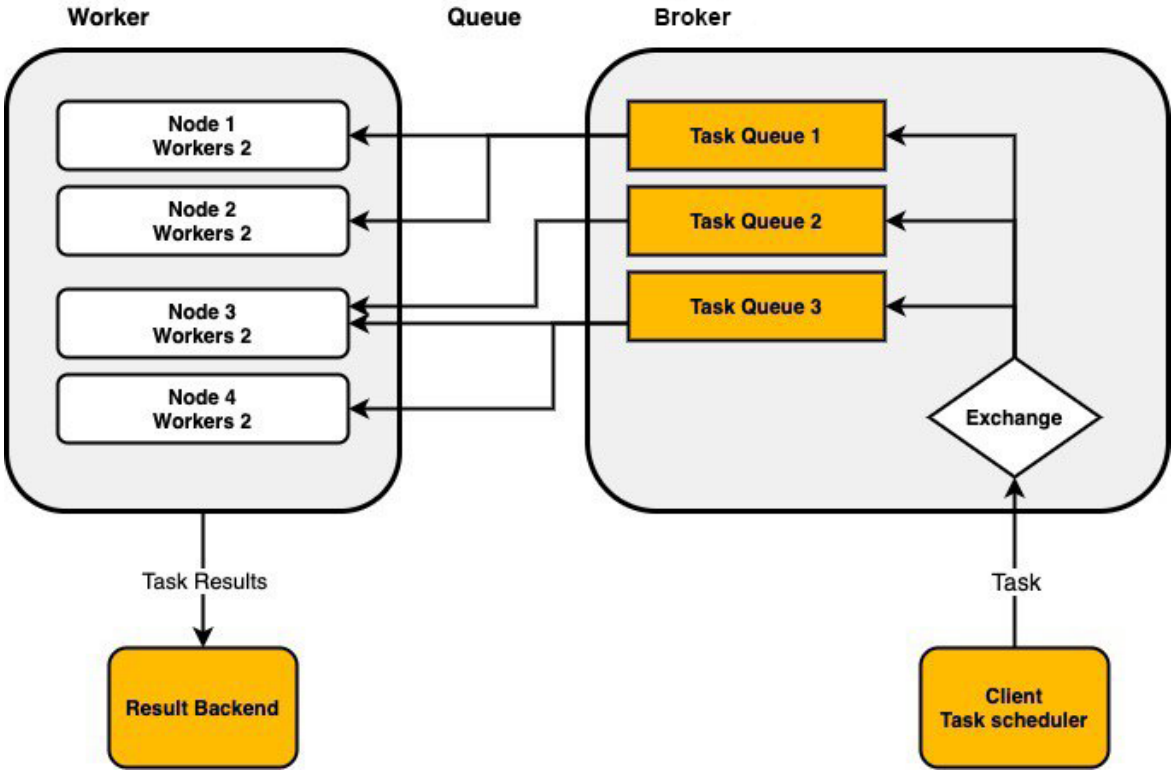


Рис. 3.5. Візуалізація взаємодії бібліотеки менеджера завдань та брокера повідомлень

3.2.1. Опис модулю планування періодичних завдань

Періодичні завдання – це такі завдання, виконання яких відбуватиметься за попередньо визначені проміжки часу. Тому в розроблюваній бібліотеці є модуль, який буде планувальником таких завдань. Цей планувальник надішле повідомлення в чергу повідомлень, коли настане час. Планувальник публікує завдання через задані проміжки часу, а потім вони виконуються наявними робочими вузлами кластеру. За графіком одночасно повинен працювати лише один планувальник, інакше завдання можуть бути поставлені повторно, що призведе до виникнення небажаних дублікатів.

Записи завдань, які повинні бути заплановані та їх графік беруться з відповідних налаштувань – schedules.

Модуль приймає на вхід конфігурацію, яка містить розклади завдань та часовий пояс. Конфігурація обов’язково повинна містити розклади завдань, інакше модуль не зможе коректно працювати. Якщо конфігурація є коректною, планувальник завдань може надсилати завдання брокеру повідомлень. Завдання обираються так: планувальник передивляється всі завдання та вибирає ті, що виконувалися останній раз рівно стільки часу назад, скільки вказано в конфігурації. Після відправки завдання на виконання, планувальник встановлює час його наступного виконання.

Модуль також підтримує поступове завершення, тобто якщо щось піде не так, або планувальник буде зупинений за допомогою cli, перед завершенням буде закрито з'єднання з брокером. Для цього додаються обробники сигналів переривання SIGTERM та SIGINT, які в свою чергу термінал відправляє в процес, коли користувач натискає ctrl-c. Ці обробники потрібні для того, щоб закрити з’єднання з RabbitMQ у випадку, коли планувальник було зупинено з командного рядка.

3.2.2. Опис модулю, що відповідає за конфігурацію періодичного завдання

Періодичні завдання, що налаштовані просто за часом, не є зручними у складних випадках використання, для таких випадків і використовується модуль Crontab.

Crontab – це графік виконання завдань, який заснований на часі, але є більш гнучким у налаштуванні. Він планує завдання, які потрібно виконувати у точно визначений час, дату або навіть інтервал часу. Запропонована реалізація має дещо спільне з Unix кроном, тому є зручною та зрозумілою [14].

Даний модуль приймає на вхід час, в який завдання повинно бути виконано, з точністю до хвилини та обчислює час наступного виконання завдання.

3.2.3. Опис модулю, який представляє завдання

Завдання – це базове поняття для бібліотеки асинхронної черги завдань і даний модуль описує інтерфейс для його використання в додатку. Клас завдання може бути створений з будь-якої звичайної Python функції. Завдання може бут виконане, як звичайна синхронна функція, може бути викликана, як корутина, або надіслана виконавцю – тоді його обробить модуль Worker, що буде розглянутий далі.

Кожне завдання має унікальне ім'я, і це ім'я прикріплене до повідомлення, щоб працівник міг знайти потрібну функцію для виконання. Ім’я повинно генеруватися автоматично і базуватися на імені функції, що лежить в основі завдання, та модулю, в якому вона визначена.

3.2.4. Опис модулю виконавця

Модуль виконавця відповідає за отримання повідомлень, виконування завдань, відправку результатів, обробку сигналів, налаштування логування тощо.

На вхід виконавець приймає конфігурацю, яка містить функції, які потрібно виконати при старті виконавця та при завершенні його роботи. Наприклад, якщо виконавцю потрібен доступ до бази даних, треба передати йому в конфігурації функцію, яка встановить з’єднання з базою даних, та функцію, яка завершить відповідне з’єднання перед завершенням роботи. Також виконавець отримує значення конкурентності, тобто значення, що встановить “prefetch count” для RabbitMQ. Це потрібно для забезпечення можливості обмежувати кількість непідтверджених повідомлень на каналі при обробці повідомлення. Якщо відповідне значення не встановлене, виконавець встановить його сам у певне

значення за замовчуванням. Виконавець обчислює кількість завдань, які надійшли йому на виконання і не візьме більше, ніж йому дозволяє встановлене значення конкурентності.

Починаючи свою роботу виконавець спочатку встановлює з’єднання з RabbitMQ, потім виконує всі функції з конфігурації, які повинен виконати, створює чергу за замовчуванням та визначає метод, що буде приймати повідомлення.

Приймаючи завдання до роботи, виконавець повинен знайти відповідну функцію за ім’ям, що міститься у повідомленні та виконати її. Якщо виконання було успішне, виконавець повинен зробити запис, який містить назву завдання та час виконання. У випадку якщо функція виконається з помилкою, буде перехоплене виключення та зроблений відповідний запис про виконання.

Виконання завдання може зайняти деякий час, тому може статися таке, що один з виконавців приступить до виконання довгого завдання і аварійно перерве свою роботу, лише частково виконавши завдання. Якщо видаляти повідомлення з пам'яті як тільки RabbitMQ доставить його споживачу, у випадку, коли виконавець несподівано завершить свою роботу, повідомлення, яке він обробляв, буде втрачене. Ми також втратимо всі повідомлення, які були надіслані цьому конкретному виконавцю, але ще не були оброблені. Втрата даних є недопустимою, тому якщо виконавець становиться недоступним повідомлення повинні бути доставлені іншому виконавцю.

Щоб переконатися, що повідомлення ніколи не втрачається, RabbitMQ підтримує підтвердження повідомлень. Від споживача повертається певний сигнал, щоб повідомити RabbitMQ, що певне повідомлення було отримано, оброблено і що RabbitMQ може видалити його.

Якщо споживач некоректно завершує свою роботу (його канал закритий, з'єднання перервано або втрачено з’єднання TCP), не

надсилаючи сигнал підтвердження, RabbitMQ зрозуміє, що повідомлення не було оброблено повністю, і повторно поставить його в чергу. Якщо в той же час є інші споживачі в Інтернеті, вони швидко передадуть їх іншому споживачеві. Таким чином ви можете бути впевнені, що жодне повідомлення не втрачається, навіть якщо працівники періодично помирають. RabbitMQ повторно доставить повідомлення, коли споживач помре. Це добре, навіть якщо обробка повідомлення займає дуже-дуже довго.

Описане вище досягається за допомогою надсилання вручну підтвердження RabbitMQ про успішне оброблення повідомлення – basic.ack. Якщо надсилання підтвердження виконається з помилкою, повідомлення буде повернуто в чергу та згодом повторно взяте до роботи. Навіть якщо виконавець буде зупинений з командної строки з ctrl-c, повідомлення не буде втрачене, завдяки цьому механізму роботи.

Після виконання виконавець зменшить кількість завдань, що зараз взяті ним для виконання та оповістить усі компоненти програми про завершення даного завдання.

Перед завершенням своєї роботи виконавець повинен переконатися, що всі взяті ним завдання завершені та виконати всі функції, які передані йому в конфігурації для виконання при завершенні роботи.

3.2.5. Опис головного модулю додатку

Даний модуль відповідає за керування всією бібліотекою та забезпечує взаємодію всіх компонентів програми.

Приймає на вхід ім’я екземпляру додатку, конфігурації та цикл подій. В конфігурації повинні бути:

- адреса підключення до RabbitMQ;
- конфігурації виконавця;
- конфігурації планувальника завдань;

- префікс для імені черги, щоб визначати, які черги належать даному екземпляру додатку;
- ім’я черги повідомлень за замовчуванням.

Цикл подій є необов’язковим аргументом, якщо його не передавати буде використаний цикл подій asyncio.

Даний модуль визначає метод, який використовується всіма іншими модулями для встановлення з'єднання з RabbitMQ. Саме тут встановлюється з’єднання, яке відновить попередній стан після підключення. Після встановлення з'єднання канали, черги та обміни з їх прив'язкою будуть відновлені. Також в цьому методі створюється канал. Даний модуль також надає можливість доступитися до об’єкту з’єднання, каналу та до черги повідомлень за замовчуванням іншим модулям. Також саме тут визначений метод закриття з’єднання.

Цей модуль повинен надавати інтерфейс реєстрації користувацької функції у додатку як завдання. Створюється екземпляр класу завдання, який був описаний вище, та записується за ім’ям завдання в стан додатку черги завдань.

Також даний модуль реалізує метод, за допомогою якого інші модулі можуть надсилати повідомлення брокеру. Далі перед надсиланням повідомлення нам потрібно переконатися в наявності черги одержувача. Якщо повідомлення буде надіслано в неіснуючу чергу, RabbitMQ просто пропустить повідомлення. У RabbitMQ повідомлення ніколи не можна надсилати безпосередньо в чергу, воно завжди проходить через точку обміну. Насправді, досить часто програма, що надсилає повідомлення, навіть не знає, чи будуть вони взагалі буде доставлені до будь-якої черги. Вона надсилає повідомлення до точки обміну, яка отримує повідомлення та надсилає їх до черг. В точці доступу повинно бути точно відомо, що робити з отриманим повідомленням – чи слід його додавати до одної або до багатьох черг, або ж слід його відкинути. Правила для цього

визначаються типом обміну. Є декілька типів обміну: default, direct, topic, headers та fanout.

default – це прямий обмін без імені, попередньо оголошеного брокером, у нього є важлива властивість, яке робить його дуже корисним для простих додатків: кожна створена черга автоматично прив’язується до обміну за допомогою ключа маршрутизації, який співпадає з назвою черги.

direct доставляє повідомлення в черги на основі ключа маршрутизації повідомлень та підходить для одноадресної маршрутизації повідомлень. Черга прив'язується до обміну за допомогою ключа маршрутизації і коли нове повідомлення з відповідним ключем надходить до точки обміну, вона спрямовує його до відповідної черги.

fanout направляє повідомлення у всі черги, які пов'язані з ним, і ключ маршрутизації ігнорується.

topic направляє повідомленнями маршрутів до однієї чи багатьох черг на основі відповідності між ключем маршрутизації повідомлення та шаблоном, який використовувався для прив’язки черги до обміну.

headers призначений для маршрутизації на декілька атрибутів, що легше передати як заголовки повідомлень, ніж ключ маршрутизації. В цьому типі обміну ігнорується ключ маршрутизації. Натомість атрибути, що використовуються для маршрутизації, беруться з атрибута заголовків. Повідомлення знаходить свою чергу, якщо значення заголовка дорівнює значенню, вказаному при прив'язці [3].

Даний модуль використовує для реалізації надсилання обмін з типом default через його простоту.

3.2.6. Опис CLI

Розроблювана бібліотека дозволяє керувати своїми модулями за допомогою інтерфейсу командного рядка або CLI. Інтерфейс повинен приймати аргументи, коректно їх обробляти та надсилати певним модулям відповідні команди на опрацювання.

Для того, щоб скористатися бібліотекою з командного рядка треба запустити додаток асинхронної черги завдань за допомогою команди tabasco та вказати шлях до об'єкту екземпляру додатку черги. Також доступні такі підкоманди як: worker, beat, tasks та delay.

- за допомогою команди worker можна запустити екземпляр виконавця та передати йому такі параметри як конкурентність або кількість повідомлень, що будуть одночасно знаходитись в каналі, та рівень логування, після чого виконавець буде запущено;
- команда beat надає можливість запустити планувальник завдань та приймає на вхід лише рівень логування;
- команда delay надсилає задання на виконання і приймає такі параметри: ім’я завдання, позиційні аргументи та аргументи за ключовими словами;
- за допомогою команди tasks можна переглянути всі зареєстровані в додатку завдання.

Також кожна команда має справку, що робить CLI більш зрозумілим для користувача.

Отже, було проведено детальний аналіз розроблюваної бібліотеки. Досліджені проблематика, цілі та результати, які потрібно досягти в ході розробки, та на основі цього сформовано ряд загальних вимог до розробки та деякі з них розглянуті більш детально. Також була описана логічна структура додатку в цілому та алгоритмічна побудова кожного модулю окремо. Визначено особливості роботи кожної частини програми в основних сценаріях, та дані, які потрібні на вхід, і що буде отримано в результаті.

4. ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ

4.1. Опис використаних інструментів, мови та сторонніх залежностей

Для створення якісного програмного забезпечення необхідно продумати його архітектуру. Архітектура програмного забезпечення описує фундаментальні структури системи і певні шаблони створення таких структур, та полягає у прийнятті певних важливих рішень, які важко буде змінити в майбутньому.

Деякі з архітектурних рішень включають в себе те, які алгоритми, шаблони проєктування та структури даних будуть використані про розробці програми, які інструменти буде використовувати система та взаємодію з ними, як буде проходити розгортання програмного забезпечення та інше.

Вибір мови програмування та інструментів таких, як бази даних, черги повідомлень та інші, теж є важливим архітектурним рішенням, адже від правильного прийняття такого рішення може залежати успіх реалізації системи в цілому.

Як основа розробленої бібліотеки був обраний asyncio – модуль стандартної бібліотеки Python для написання конкурентного коду за допомогою синтаксису async/await, що використовується як основа для безлічі асинхронних фреймворків Python, які забезпечують високопродуктивну роботу з мережею та веб-серверами, бібліотек підключення до бази даних, розподілених черг завдань тощо. Цей модуль також дуже добре підходить для додатків, в яких багато операцій вводу-виводу та структурованого мережевого коду високого рівня.

Ядром кожної програми на asyncio є цикл подій, який виконує асинхронні завдання, зворотні виклики, операції з мережевого вводу-виводу та виконують підпроцеси. Він надає API для:

- планування викликів;

- передача даних по мережі;
- виконання запитів DNS;
- обробка сигналів ОС;
- зручні абстракції для створення веб-серверів та з'єднань;
- робота з підпроцесами асинхронно.

По суті, все що робить цикл подій – це чекає, коли події відбуватимуться перед тим, як співпіставити кожну подію функції, яку ми явно зв’язали із вказаним типом події.

Додатки розроблені на asyncio є значно швидшими за синхронні додатки на Python, але існує спосіб зробити asyncio ще швидшим. Для розроблення даної бібліотеки був використаний цикл подій uvloop, який є швидшою альтернативою вбудованого циклу подій asyncio. Згідно з документацією uvloop принаймні в 2 рази швидший, ніж nodejs, gevent, і будь-який асинхронний фреймворк Python. Продуктивність asyncio на основі uvloop близька до ефективності програм Golang. uvloop написаний на CPython і побудований поверх libuv – високопродуктивої, багатоплатформної асинхронної бібліотеки, яку також використовує nodejs.

В якості брокеру повідомлень був обраний RabbitMQ, який на думку автора найкраще підходить для розроблення саме легкої, простої у використанні, але одночасно і потужної бібліотеки. RabbitMQ має багато можливостей для масштабування, забезпечення відмовостійкості та збереження даних у разі, якщо відмова таки сталася. Також він надає можливості моніторингу, досить швидкого розгортання та навколо нього велика спільнота розробників, і як наслідок є готові плагіни та клієнтські бібліотеки. Одною з таких бібліотек є aio-pika, що в свою чергу є обгорткою над більш низькорівневою бібліотекою aiormq.

Розглянемо детальніше бібліотеки aio-pika та aiormq. Бібліотека aiormq надає такі можливості:

- підключення за URL-адресою;

- буферна черга для отриманих даних;
- механізм аутентифікації PLAIN – який працює за замовчуванням на сервері та клієнтах RabbitMQ;
- підтримка транзакцій;
- відстеження немаршрутизуємих повідомлень;
- повна підтримка SSL/TLS;
- анотації типів Python;
- властивість “Підтвердження видавцеві» (Publisher Confirms), яке є розширенням наявної специфікації AMQP і підтримується тільки клієнтськими бібліотеками RabbitMQ – перш ніж публікувати будь-яке повідомлення, видавець повідомлення повинен випустити запит RPC Confirm.Select в сторону RabbitMQ і очікувати відгуку Confirm.SelectOk, щоб бути впевненим, що підтвердження доставки увімкнені, з цього моменту для кожного повідомлення, яке видавець відправить до RabbitMQ, його сервер буде відповідати підтверджуючим відгуком (Basic.Ack) або запереченням (Basic.Nack) (рис 4.1).

Таким чином, можна вважати, що бібліотека aiormq є хорошою базою для розроблення бібліотеки за визначеними вимогами, адже вона забезпечує відмовостійкість реалізуючи “Підтвердження видавцеві” в RabbitMQ, підтримуючи анотації типів та працюючи з asyncio.

Далі розглянемо більш високорівневу бібліотеку – aio-pika. Бібіліотека aio-pika – це більш високорівнева обгортка над aiormq для asyncio. Головними її перевагами є:

- повністю асинхронний API;
- об'єктно-орієнтований API;
- автоматично відновлюване з'єднання з повним відновленням стану за допомогою методу connect_robust;
- сумісний з Python 3.4+;
- зручна підтримка транзакцій;

- зрозуміла імплементація властивості “Підтвердження видавцеві”.

Засоби реалізації були підібрані з урахуванням вимог до розроблюваної бібліотеки, а саме – всі сторонні бібліотеки працюють с asyncio, мають підтримку анотацій типів, мають спільноту розробників та підтримуються ними.

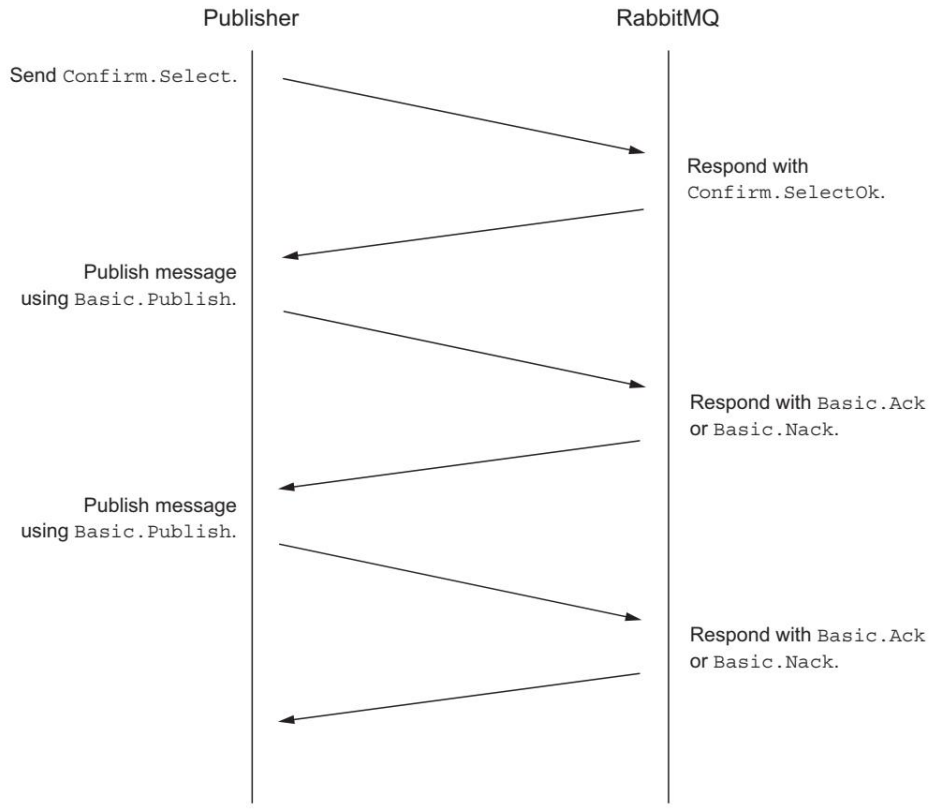


Рис. 4.1. Послідовність повідомлень, що відправляються до та від RabbitMQ для доставки підтверджень

4.2. Аспекти реалізації структурно-алгоритмічної організації бібліотеки

Докладно розглянемо структуру розроблюваної бібліотеки, яка отримала робоче ім'я Tabasco.

Бібліотека Tabasco складається з таких Python модулів:

- __init__.py – використовується для позначення каталогів на диску як каталоги пакетів Python, в даному випадку в цьому

файлі перераховані імена, що визначені в даному модулі, які слід імпортувати, коли використовується from tabasco import *;

- beat.py – містить клас TabacsoBeat, що описує планувальник періодичних завдань;
- config.py – містить класи, які описують конфігурації, що будуть передані іншим класам, містить класи TabascoWorkerConfig, TabascoBeatConfigBase, TabascoBeatConfig, TabascoBeatSchedule, та загальний TabascoConfig;
- crontab.py – містить клас TabascoCrontab, що описує графік виконання періодичних завдань, який потім передається до TabascoBeat;
- tabasco.py – містить клас Tabasco, який описує додаток асинхронної черги завдань в цілому та зв’язує між собою різні компоненти бібліотеки;
- task.py – містить клас TabascoTask, що описує саме завдання, яке повинно бути передане виконавцеві;
- worker.py – містить клас, що описує виконавця завдань.

Також бібліотека містить пакет CLI, в якому знаходяться такі модулі:

- beat.py – містить функцію prepare_beat_parser, що аналізує аргументи командного рядка, які були передані команді beat, та функцію beat_action, яка запускає саме планувальник завдань;
- delay.py – містить функцію prepare_delay_parser, що аналізує аргументи, які були передані команді delay; приватну функцію _async_delay_action, яка формує сигнатуру завдання та відправляє його в чергу на виконання; публічну функцію delay_action, яка збирає аргументи командного рядку та готує до передачі в структуру TabascoTaskSignature, що представляє сигнатуру завдання, а також логує успішну передачу завдання до черги;

- main.py – містить функцію main, яка відповідає за аналіз аргументів головної команди командного рядку даної бібліотеки – tabasco; реєструє функції-аналізатори дочірніх команд таких як: worker, beat, tasks, delay;
- tasks.py – містить функцію tasks_action, яка виводить список імен усіх зареєстрованих у додатку завдань;
- utils.py – містить функції-помічники такі як: функція, що витягує екземпляр додатку Tabasco з аргументів командного рядка, в яких передається шлях до потрібного модуля, та функція, що ініціалізує логування;
- worker.py – містить функцію prepare_worker_parser, що аналізує аргументи командного рядка, які були передані команді worker, та функцію worker_action, яка запускає виконавець завдань.

Загальну структуру бібліотеки можна побачити на рис. 4.2.

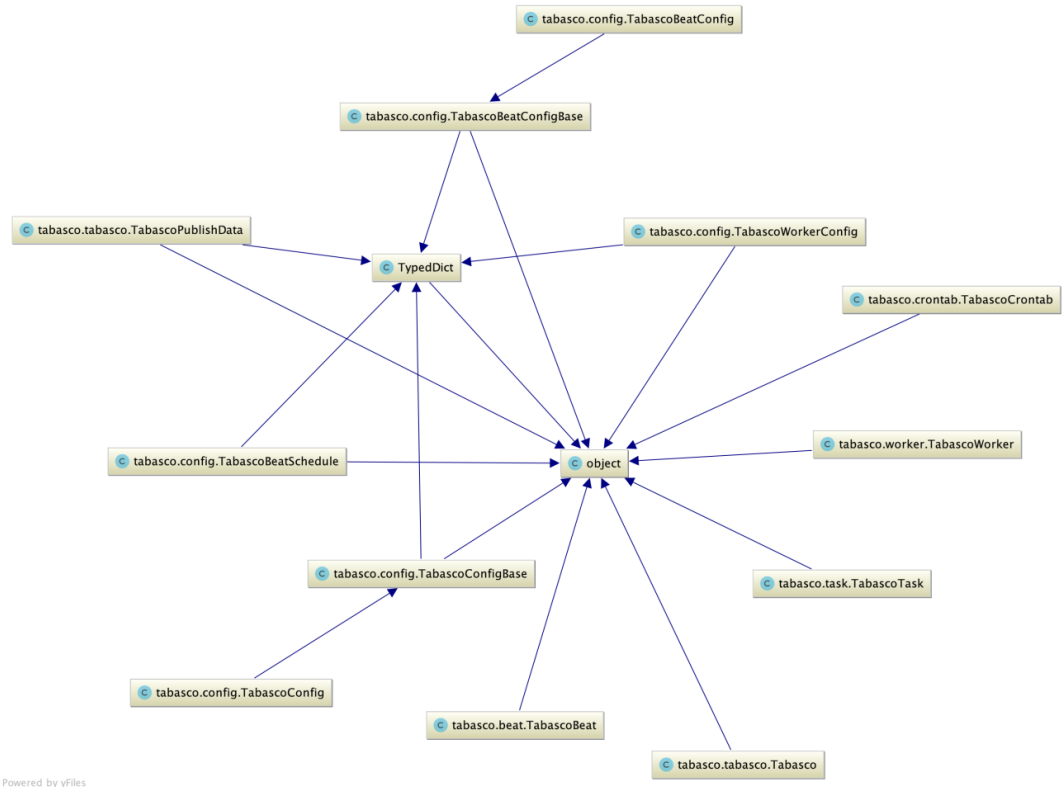


Рис. 4.2. Основні класи бібліотеки Tabasco та їх зв'яки з деякими стандартними класами

4.2.1. Опис головного модулю бібліотеки

Модуль tabasco є головним модулем бібліотеки і містить в собі 2 класи: Tabasco та TabascoPublishData.

Клас Tabasco є головним класом, яких зв’язує в собі всі інші компоненти бібліотеки. Цей клас має наступні методи:

- __init__ – цей метод є конструктором класу та приймає як аргументи ім’я та конфігурації додатку у вигляду екземпляру класу TabascoConfig, звідки отримує параметри URL для підключення до RabbitMQ, префікс імені черги повідомлень, ім’я черги за замовчуванням та об’єкт циклу подій, також визначаються пусті поля _connection, _channel та tasks, які потрібні для зберігання об’єктів з’єднання з брокером, каналу в RabbitMQ та списку об'єктів типу TabascoTask – зареєстрованих в додатку завдань;
- connect – за допомогою цього методу можна встановити з’єднання з RabbitMQ, його використовують інші частини бібліотеки, наприклад, планувальник завдань та виконавець;
- disconnect – цей метод реалізує коректне завершення з’єднання, його також використовують інші модулі;
- default_amqp_queue_name – даний метод повертає ім’я черги повідомлень, що використовує додаток, ім’я складається з префіксу та імені черги за замовчуванням, які вказані у TabascoConfig;
- task – декоратор, який приймає на вхід будь-яку Python функцію, робить з неї об’єкт типу TabascoTask та реєструє як завдання в екземплярі додатку Tabasco;
- publish – цей метод приймає сигнатуру завдання, що є об’єктом типу TabascoTaskSignature, який містить ім'я завдання, позиційовані та іменні аргументи, присвоює завданню унікальний ідентифікатор та надсилає в канал RabbitMQ в

точку обміну за замовчуванням з ключем маршрутизації, який був згенерований методом default_amqp_queue_name.

Також даний клас має дві властивості: connection та channel, які потрібні для доступу до приватних полів _connection та _channel відповідно.

4.2.2. Опис модулю планувальника періодичних завдань

Модуль beat містить клас TabascoBeat, який реалізує функціональність планувальника періодичних завдань.

В цьому класі визначені такі поля як:

- app – екземпляр додатку – об’єкт типу Tabasco, який містить всі загальні методи, поля та стан черги завдань;
- timezone – налаштування часового поясу;
- schedules – налаштування графіку виконання завдань;
- _period_schedules – поле, яке зберігає графіки виконання завдань у вигляді часових проміжків;
- _crontab_schedules – поле, яке зберігає графіки виконання завдань у вигляді об’єктів TabascoCrontab.

Також даний клас має такі методи:

- __init__ – приймає в якості аргумента екземпляр додатку – об’єкт типу Tabasco, та визначає всі потрібні поля з поля config цього об'єкту;
- _put_crontab_schedule – цей метод встановлює час наступного виконання завдання;
- _publish_task – збирає сигнатуру завдання та публікує його використовуючи метод publish класу Tabasco;
- _beat – обирає завдання, яке треба виконати згідно з графіком, з полів _period_schedules та _crontab_schedules і планує наступне виконання за допомогою методу _put_crontab_schedule;

- _main – викликає метод _beat кожну секунду та орієнтується на флаг _should_stop для завершення роботи планувальника;
- _startup – виконується при запуску планувальника, встановлює з'єднання з RabbitMQ за допомогою методу connect класу Tabasco та викликає метод _main;
- _cleanup – виконується перед завершенням роботи планувальника, розриває з’єднання з RabbitMQ, зупиняє цикл подій;
- _close – прибирає обробники сигналів SIGTERM та SIGINT, викликає метод _cleanup;
- run – встановлює обробники сигналів SIGTERM та SIGINT, викликає метод _startup.

4.2.3. Опис модулю налаштування графіку виконання періодичних завдань

Модуль crontab дозволяє налаштовувати графік виконання періодичних завдань. Модуль окрім стандартного методу __init__ має лише один метод – next_timestamp, який приймає поточний час та повертає час наступного виконання завдання. Використовується для налаштування часу виконання, наприклад, в конкретну секунду кожної хвилини, або конкретну годину кожного дня. Це може бути задано за допомогою відповідних параметрів, таких як: day, hour, minute та second. Реалізація цього модулю схожа на cron завдання, що повинно достатньо покривати потреби у налаштуваннях часу виконання завдань.

4.2.4. Опис модулю виконавця завдань

Модуль, що називається worker, потрібен для того щоб забирати завдання з черги для подальшого їх виконання. Цей модуль приймає на вхід додаток черги завдань – об’єкт типу Tabasco та параметр concurrency,

що визначає кількість повідомлень, що одночасно можуть знаходитися в каналі RabbitMQ, з яким працює виконавець.

Даний модуль визначає такі методи, як:

- _on_message – метод, що приймає повідомлення, яке надійшло в канал, з яким працює черга, знаходить у повідомленні ім’я завдання та по знайденому імені викликає Python функцію, що зареєстрована в додатку черги під відповідним ім’ям;
- _startup – виконується при запуску виконувача, встановлює з'єднання з RabbitMQ та викликає всі методи, що задані в конфігураціях і повинні бути виконані на старті; встановлює автоматично відновлюване з’єднання з брокером, а саме з конкретним каналом с заданим параметром concurrency; визначає чергу, з якою буде працювати, та встановлює функцію _on_message як обробник надходження повідомлення в дану чергу;
- _cleanup – виконується перед завершенням роботи виконавця, розриває з’єднання з RabbitMQ, очікує доки всі взяті на виконання завдання будуть виконані, для коректного завершення роботи виконавця;
- _close та run – аналогічні однойменним методам планувальника завдань.

4.2.5. Опис модулю завдання

Даний модуль називається task і містить всього два класи – TabascoTask та TabascoTaskSignature. Клас TabascoTask містить такі методи:

- __init__.py – стандартний метод ініціалізації, що приймає об’єкт типу Tabasco, та функцію, яку треба буде виконати у якості завдання, також визначає поле класу name – ім’я

завдання, що складається с назви модулю виконуваної функції та назви самої функції;

- __call__ – викликає передану функцію з заданими аргументами;
- signature – метод, що повертає об’єкт типу TabascoTaskSignature, який містить ім'я завдання та аргументи, з якими воно повинно бути виконане;
- delay – відправляє в чергу повідомлення з даними, що повернув метод signature.

Більш детальна структура бібліотеки з багатьма описаними вище модулями представлена на рис. 4.3.

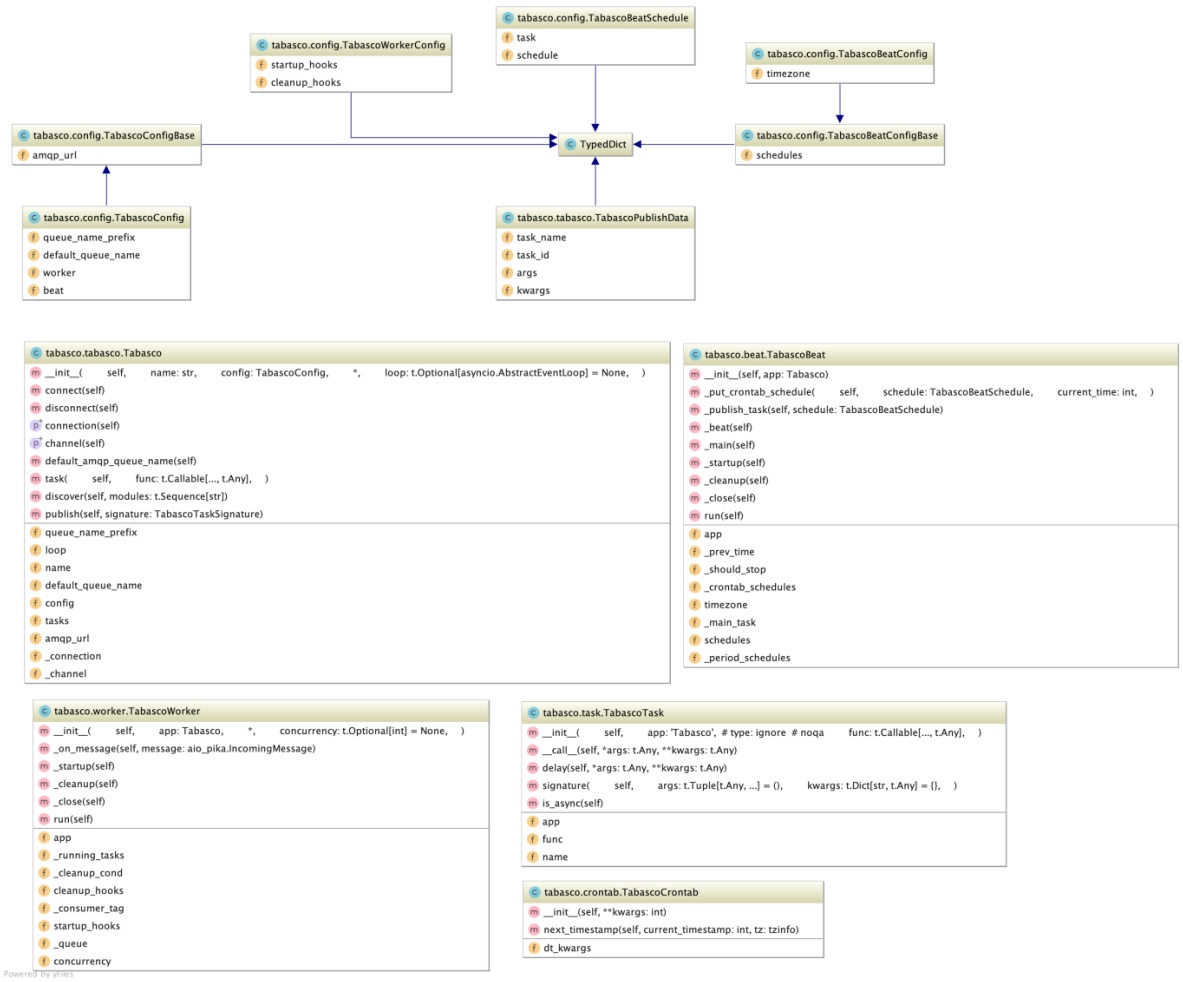


Рис. 4.3. Основні класи, методи та поля бібліотеки Tabasco

Отже, в даному підрозділі наведено опис основних модулів розроблюваної бібліотеки та визначено зв’язки, які модулі мають між собою. Також розглянуто деякі важливі деталі реалізації даної бібліотеки та показано як саме було виконано вимоги до розроблення даного програмного забезпечення.

4.3. Приклади використання бібліотеки

В даному розділі розглянуто приклади використання розробленої бібліотеки та її інтеграції в інший проєкт.

Перед використанням бібліотеки треба створити екземпляр спеціального класу, що представляє бібліотеку в цілому. Цей екземпляр є безпечним для потоків, тому в одному просторі процесу можуть спільно існувати кілька додатків Tabasco з різною конфігурацією, компонентами та завданнями. Для створення екземпляру додатку потрібно передати цього конструктору ім'я додатку, конфігурації та об'єкт циклу подій.

За допомогою спеціального методу discover можна знайти всі функції, що будуть зареєстровані в додатку як завдання. Нижче наведений приклад ініціалізації додатку Tabasco.

Приклад створення екземпляру додатку бібліотеки

```
import asyncio
from pytz import timezone
# … some more imports here

tabasco_config = TabascoConfig(
    amqp_url=config['amqp']['url'],
    worker=TabascoWorkerConfig(
        startup_hooks=[],
        cleanup_hooks=[],
    ),
    beat=TabascoBeatConfig(
        timezone=timezone.tz,
        schedules=[
            TabascoBeatSchedule(
                task=(
                    'habanero.tasks.examples_tasks.'
                    'list_scoville_table_lines'
                ),
                schedule=TabascoCrontab(microsecond=2),
            ),
        ],
```

```
    )
)

app = Tabasco('habanero', tabasco_config, loop=asyncio.get_event_loop())

app.discover(['habanero.tasks.examples_tasks'])
```

В даному прикладі додатку було передано конфігурації з посиланням для підключення до RabbitMQ, налаштуваннями виконавця, в даному випадку налаштування мінімальні, та налаштуваннями планувальника. В налаштуваннях планувальника в цьому прикладі налаштування часового поясу і лише одне періодичне завдання, яке виконується кожну секунду на другій мілісекунді.

Для того, щоб зареєструвати Python функцію як завдання у додатку бібліотеки треба обернути її в Python-декоратор @app.task.

Приклад створення завдання

```
from datetime import datetime

from habanero.tabasco import app as tabasco_app


@tabasco_app.task
async def answer_to_ultimate_question_of_life(answer: str = '42') -> str:
    return f'Answer to the Ultimate Question of Life is {answer}'


@tabasco_app.task
async def list_scoville_table_lines() -> str:
    return f'Got {datetime.now().timestamp()} Scoville heat units now!'
```

Для роботи бібліотеки потрібно запустити RabbitMQ та виконавця, який буде приймати та виконувати завдання та надсилати йому завдання за допомогою CLI або іншого Python проєкту.

Приклад успішного виконання наведеного вище завдання answer_to_ultimate_question_of_life виглядає так, як показано на рис. 4.4.

На рис. 4.4. видно як виконавець отримав завдання – “Received task …” та виконав його і повернув результат – “Succeeded task …”. Дане завдання було відправлене з невеликого асинхронного Python додатку за допомогою синтаксису delay.

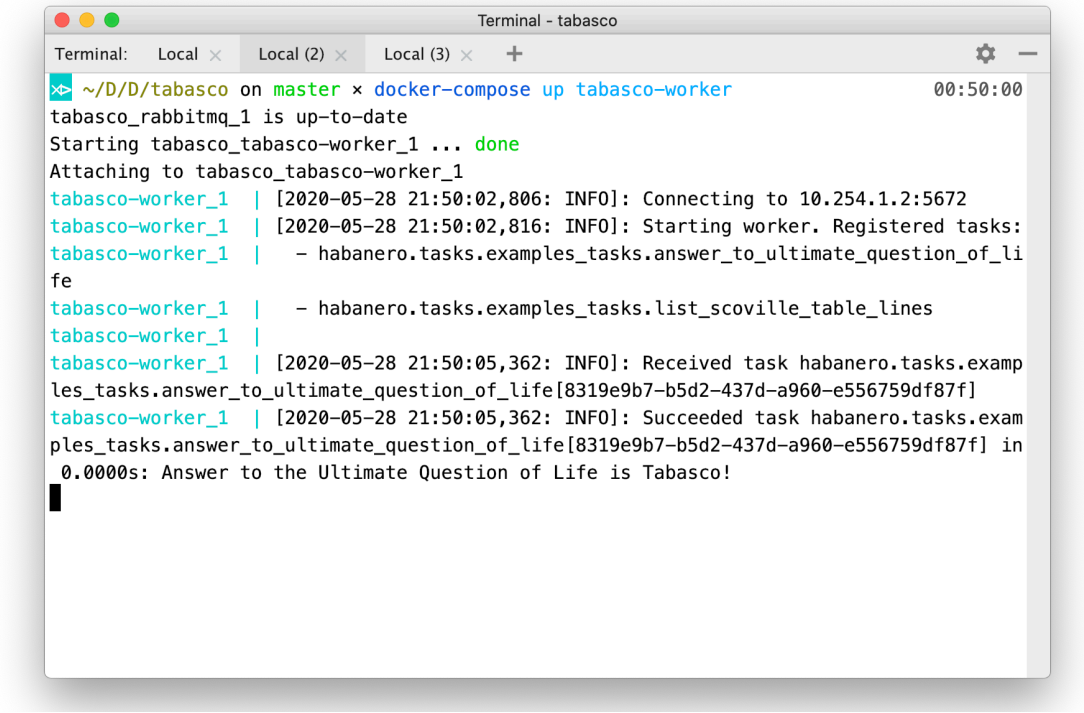


Рис. 4.4 Приклад успішного виконання завдання Tabasco

Приклад виклику завдання за допомогою синтаксису delay

```
# Direct call job as function:
result = await answer_to_ultimate_question_of_life()

# Celery-like add task to queue:
await answer_to_ultimate_question_of_life.delay(answer='Tabasco!')
```

Завдання також може бути поставлене на виконання планувальником завдань. Приклад відправлення планувальником завдання щосекунди наведений на рис. 4.5.

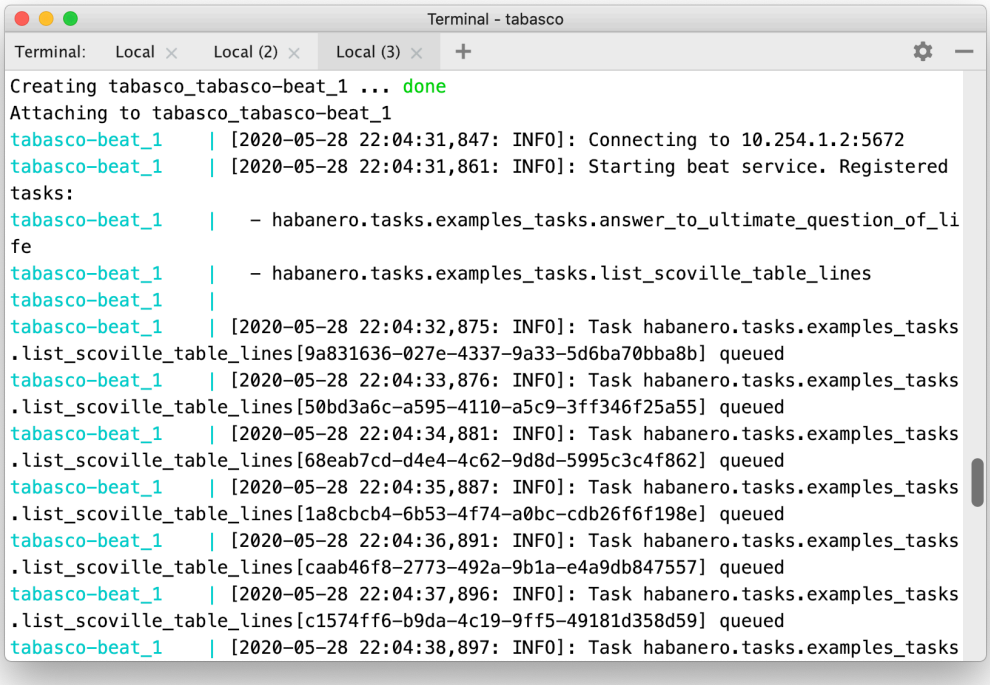


Рис. 4.5. Планувальник завдань відправляє завдання в чергу щосекунди

Графік завдання наведене в прикладі на рис. 4.5. налаштовано на виконання кожної секунди, що також можна побачити на рисунку.

В свою чергу виконавець приймає завдання та виконує їх, як і в попередньому прикладі. Результат отриманих завдань обчислюється, а про те, що це саме ті завдання, щ обули надіслані свідчить унікальний ідентифікатор завдання, який можна знайти в записах як планувальника при відправці, так і виконавця при обчисленні результату. Виведення результату обчислення виконавцем отриманих завдань можна побачити на рис. 4.6.

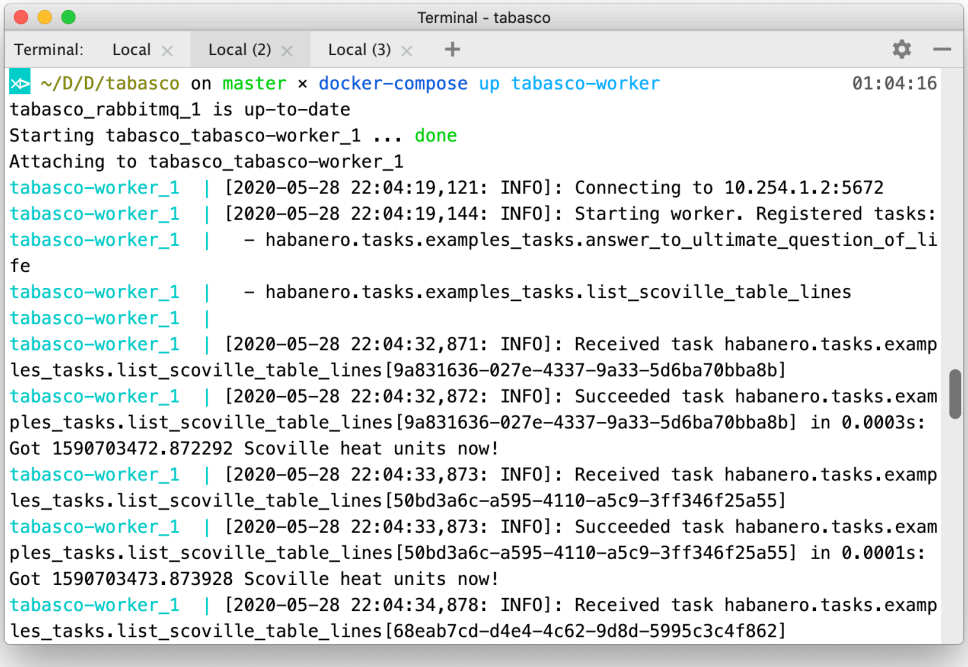


Рис. 4.6. Журналювання обчислення результатів завдань надісланих планувальником

Бібліотека також може використовуватися як утиліта командного рядка та має так команди як tasks, delay, worker та beat. Всі вони було докладно описані в пункті 3.2.6. Приклади використання деяких з них наведені на рис. 4.7. Можна побачити, що команда tasks повернула імена всіх зареєстрованих в додатку завдань, команда delay відправила в чергу завдання, а командою beat було запущено планувальник завдань.

Також на цьому рисунку можна побачити як працює коректне завершення роботи, яке реалізоване у виконавця та планувальника і описане у попередніх розділах. Планувальник спочатку закриває канал з RabbitMQ, потім закриває з’єднання і тільки тоді завершує свою роботу.

У виконавця виконує ті ж операції, а окрім цього очікує доки виконаються усі взяті ним завдання.

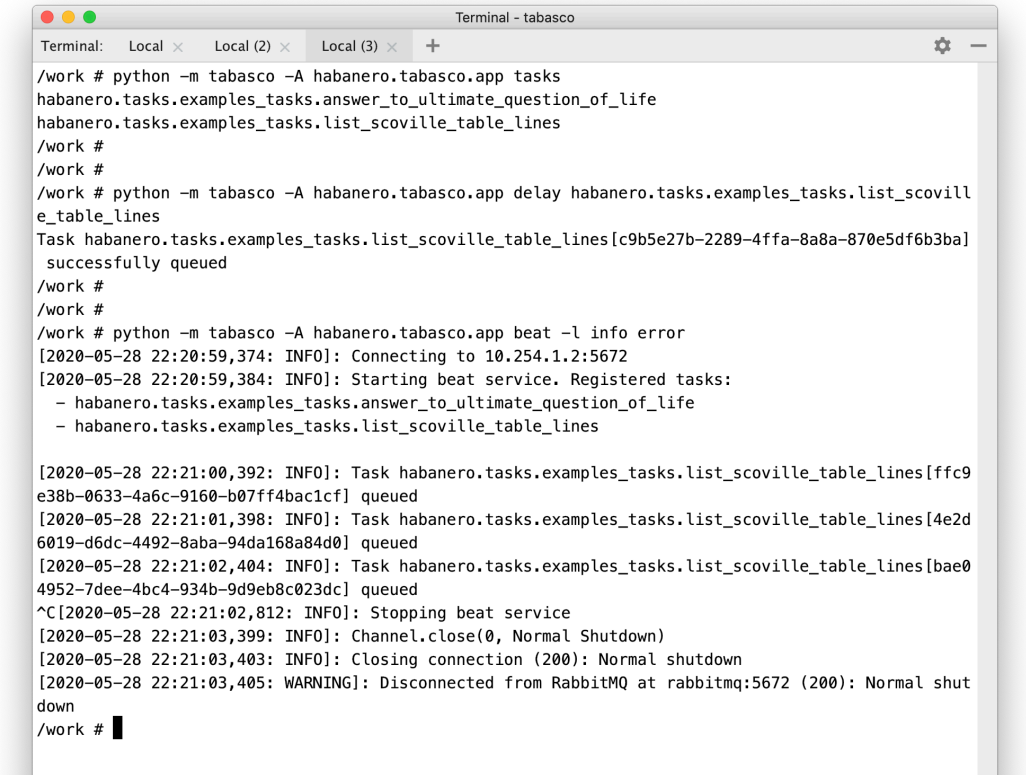


Рис. 4.7. Приклад виконання CLI команд Tabasco

4.4. Опис використаного інструментального програмного забезпечення

В ході розроблення бібліотеки Tabasco було використано інструментальні програмні засоби для контейнеризації програмних додатків, інструмент для забезпечення дотримання стилю написання коду та інструмент для статичної перевірки типів для Python 3.

Розглянемо програмний засіб для контейнеризації програмних додатків. Docker – це програмне забезпечення, що має модель платформа як послуга, і використовується для управління ізольованими Linux-контейнерами. Контейнери можуть спілкуватися між собою за допомогою спеціально налаштованих каналів. Docker дозволяє не беручи до уваги специфіку додатку, що знаходиться в контейнері, розгортати та переносити розроблюваний додаток з одного середовища в інше. Маючи спеціальні конфігурації, які знаходяться в Dockerfile, Docker сам створює

контейнери з усіма необхідними залежностями. Також існує багато все готових образів контейнерів, які можна розширювати під власні потреби.

При розробленні бібліотеки Tabasco для розгортання додатку виконавця та планувальника завдань був побудований контейнер на базі образу python-alpine та встановленими в контейнер необхідними залежностями. Для розгортання RabbitMQ був використаний готовий образ rabbitmq:3.7.8-management-alpine.

Для зручності розроблення був використаний docker-compose – це інструмент для визначення та запуску багатоконтейнерних програм Docker. Для використання docker-compose треба написати yaml-файл, у якому визначити та налаштувати всі необхідні сервіси. Після цього можна легко запустити будь-який сервіс за допомогою простої команди або ж запустити відразу всі сервіси.

Для забезпечення дотримання єдиного стилю написання коду розробниками Python був створений PEP 8, а для перевірки коду на відповідність цим рекомендаціям потрібен інструмент flake8. Він сканує код проєкту та знаходить в ньому стилістичні помилки та порушення конвенцій коду Python, і вміє працювати не тільки з PEP 8, а й з іншими правилами.

Бібліотека Tabasco розроблена з використанням flake8 та відповідає конвенціям, що викладені у PEP 8.

Також у вимогах до розроблення цієї бібліотеки було покриття коду анотаціями типів. Для перевірки правильності покриття було використано Муру – інструмент додаткової статичної перевірки типів для Python, який має на меті поєднання переваг динамічної та статичної типізації.

Отже, в даному розділі було проаналізовано структуру розроблюваної бібліотеки та використання сторонніх залежностей необхідних для її роботи. Розглянуто як використовувати та інтегрувати Tabasco до іншого проєкту, а також було розглянуто використання додаткових інструментів для розроблення та розгортання бібліотеки.

ВИСНОВКИ

В даному дипломному проєкті досягнуто поставлену ціль розроблення асинхронної черги завдань для модулю стандартної бібліотеки asyncio Python. Було проаналізовано проблеми, які повинні бути вирішені даним програмним забезпеченням, відповідно до проаналізованих проблем поставлено цілі та задачу на розроблення. Тоді було розглянуто існуючі програмні рішення та проаналізовано їх за певними критеріями. Дане дослідження показало, що не одне з існуючих рішень не відповідає висунутим вимогам, тому розроблення даного програмного забезпечення є актуальним. Після цього було обґрунтовано вибір версії мови програмування Python та вибір найбільш підходящого для вирішення поставленої задачі брокеру повідомлень. Аналіз було проведено на основі вибору інструментів такого типу для аналогічних за структурою та специфікою проєктів. Підсумовуючи останні розділи пояснювальної записки даного проєкту, було створено підхід до вирішення даної задачі та описано вимоги до створюваного рішення, розроблено спеціальні алгоритми для їх роботи та описано найважливіші моменти реалізації, показано як дане рішення може бути використане та інтегроване в інший проєкт, обрано додаткові програмні засоби для розроблення та розгортання даної бібліотеки.

Розроблення бібліотек для асинхронного Python є важливим та перспективним напрямком. У подальшому планується розвиток даної бібліотеки, а саме, запланована більш складна та гнучка реалізація парсеру налаштування графіку виконання періодичних завдань, можливість задавати автоматичне повторне виконання в разі неможливості виконання завдання з першого разу та можливість налаштування багатьох черг виконання завдань. Також планується створення повноцінного вбудованого інструменту моніторингу виконання завдань та ступеню завантаження черги.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. PEP 3156 — Asynchronous IO Support Rebooted: the "asyncio" Module [Електронний ресурс]. — Режим доступу: https://www.python.org/dev/peps/pep-3156. — (01.04.2020) — Назва з екрана.
2. asyncio — Asynchronous I/O [Електронний ресурс]. — Режим доступу: https://docs.python.org/3/library/asyncio.html. — (02.04.2020) — Назва з екрана.
3. Messaging that just works — RabbitMQ [Електронний ресурс]. — Режим доступу: https://www.rabbitmq.com. — (15.04.2020) — Назва з екрана.
4. Faust — Python Stream Processing [Електронний ресурс]. — Режим доступу: https://faust.readthedocs.io/en/latest. — (16.04.2020) — Назва з екрана.
5. Introducing Faust [Електронний ресурс]. — Режим доступу: https://faust.readthedocs.io/en/latest/introduction.html?highlight=RocksDB#what-do-i-need. — (20.04.2020) — Назва з екрана.
6. Celery — Distributed Task Queue [Електронний ресурс]. — Режим доступу: https://docs.celeryproject.org/en/stable. — (22.04.2020) — Назва з екрана.
7. Dramatiq: background tasks [Електронний ресурс]. — Режим доступу: https://dramatiq.io. — (22.04.2020) — Назва з екрана.
8. RQ — Easy jobs queues for Python [Електронний ресурс]. — Режим доступу: https://python-rq.org. — (22.04.2020) — Назва з екрана.
9. What’s New In Python 3.8 [Електронний ресурс]. — Режим доступу: https://docs.python.org/3/whatsnew/3.8.html. — (01.05.2020) — Назва з екрана.
10. Kafka 2.5 Documentation [Електронний ресурс]. — Режим доступу: https://kafka.apache.org/documentation. — (01.05.2020) — Назва з екрана.

11. Redis [Електронний ресурс]. — Режим доступу: https://redis.io. — (02.05.2020) — Назва з екрана.
12. IEEE Standart Glossary of Software Engineering Terminology [Електронний ресурс]. — Режим доступу: http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf. — (10.05.2020) — Назва з екрана.
13. Master-Worker Pattern [Електронний ресурс]. — Режим доступу: https://docs.gigaspaces.com/solution-hub/master-worker-pattern.html.— (10.05.2020) — Назва з екрана.
14. crontab (5) [Електронний ресурс]. — Режим доступу: http://www.opennet.ru/man.shtml?topic=crontab&category=5. — (10.05.2020) — Назва з екрана.

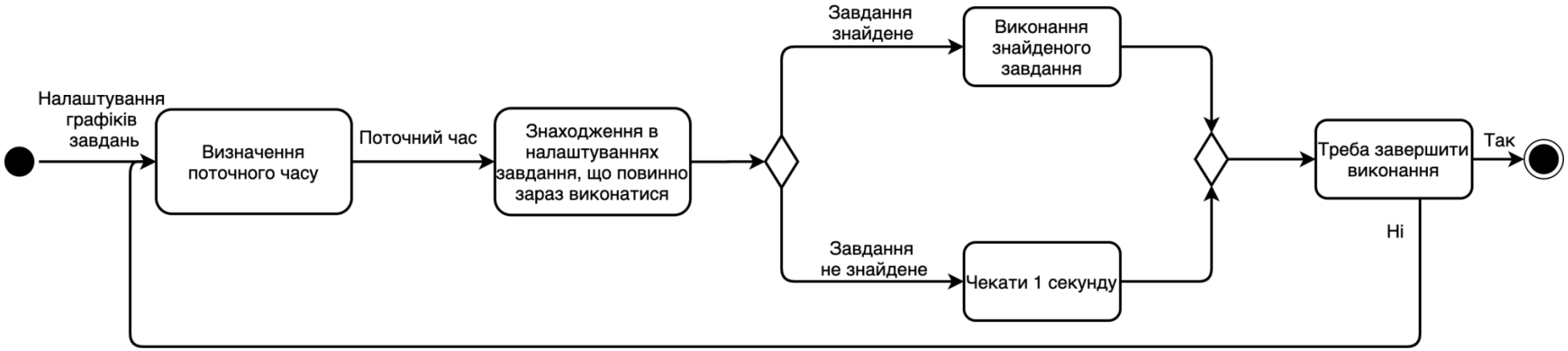
ДОДАТКИ

Додаток 1

Копії графічних матеріалів

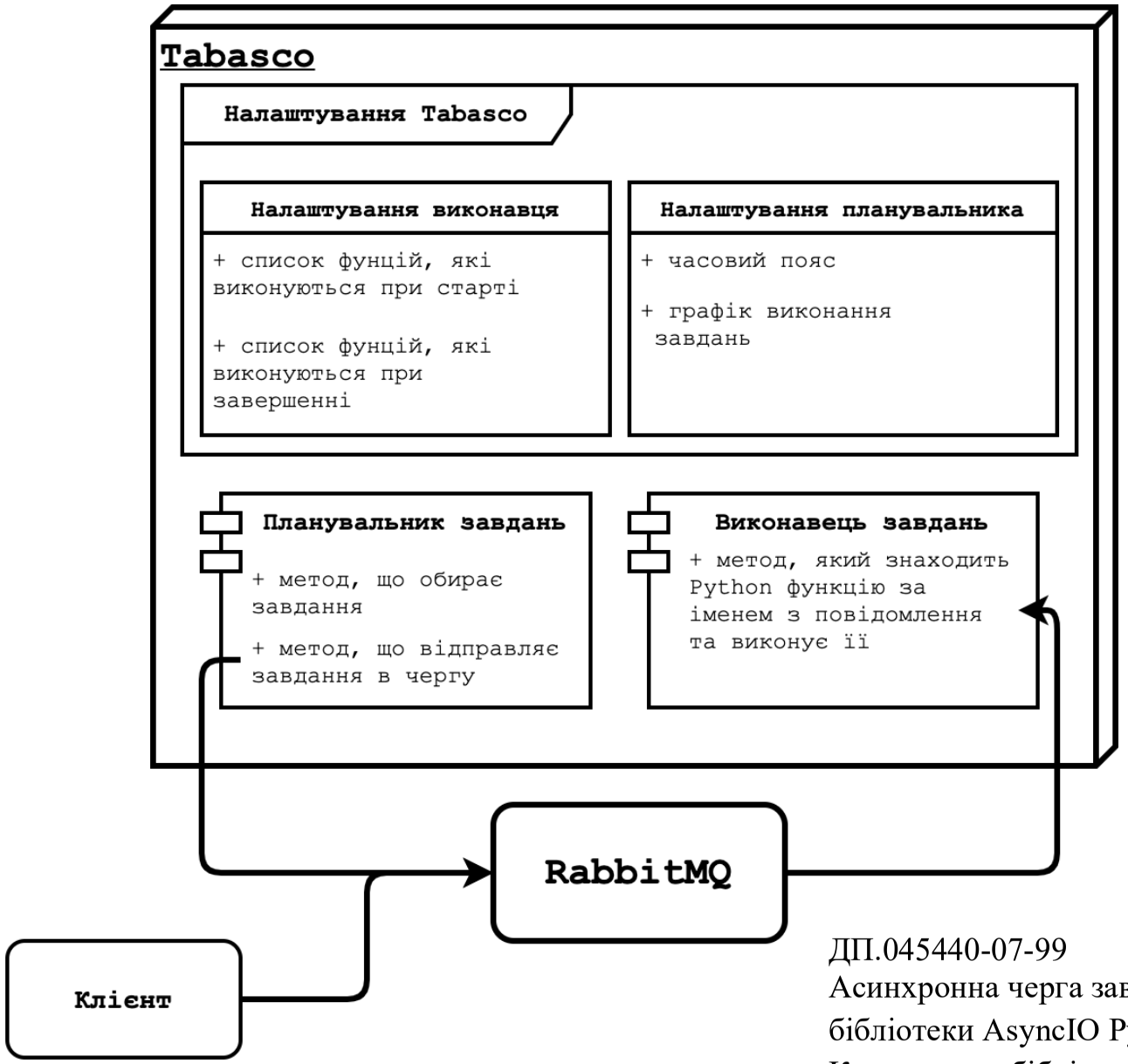
Алгоритм роботи виконавця завдань

Алгоритм роботи планувальника завдань



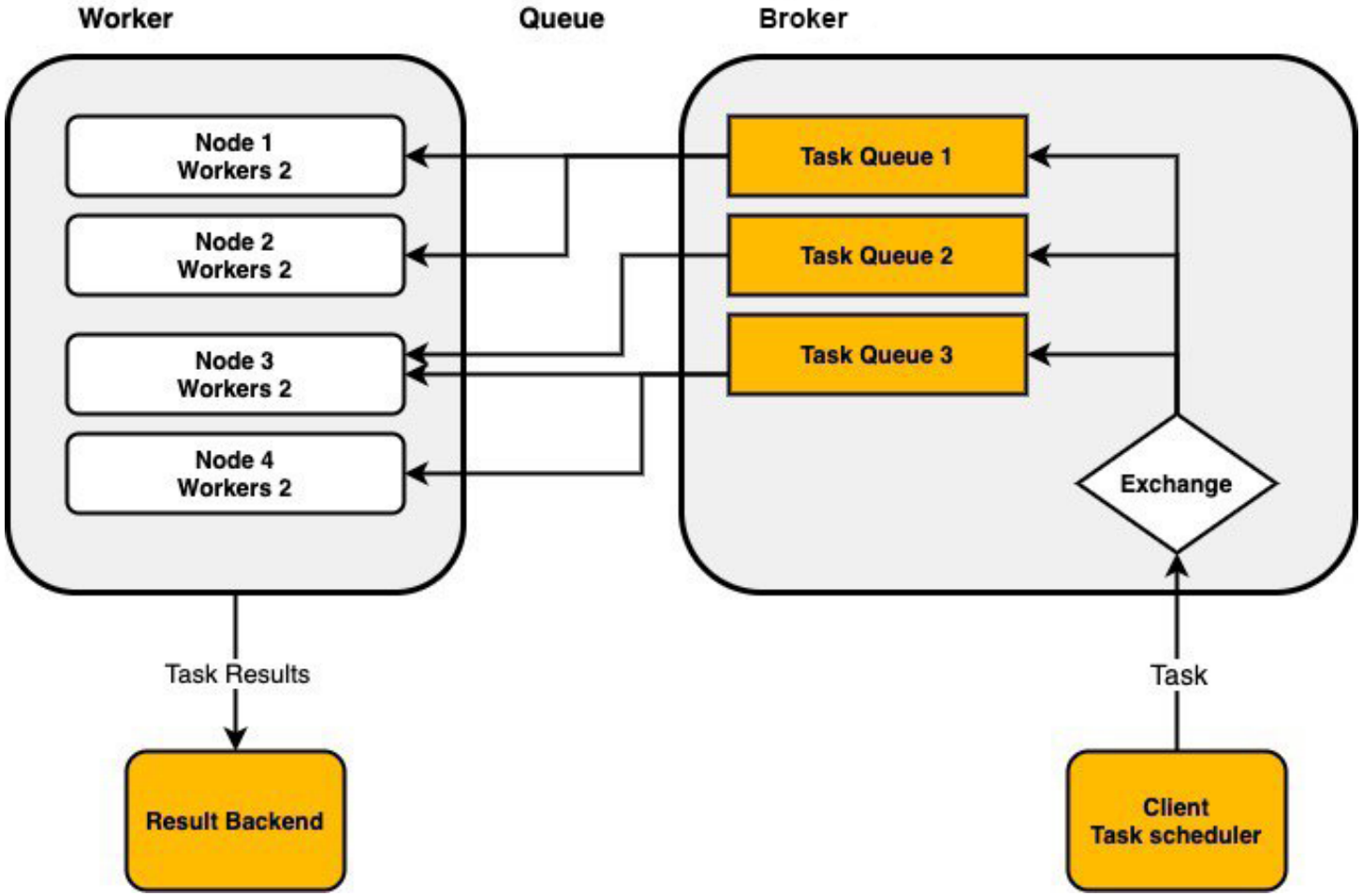
ДП.045440-06-99

Асинхронна черга завдань для бібліотеки AsyncIO Python. Алгоритми роботи виконавця та планувальника завдань. UML-діаграма діяльності



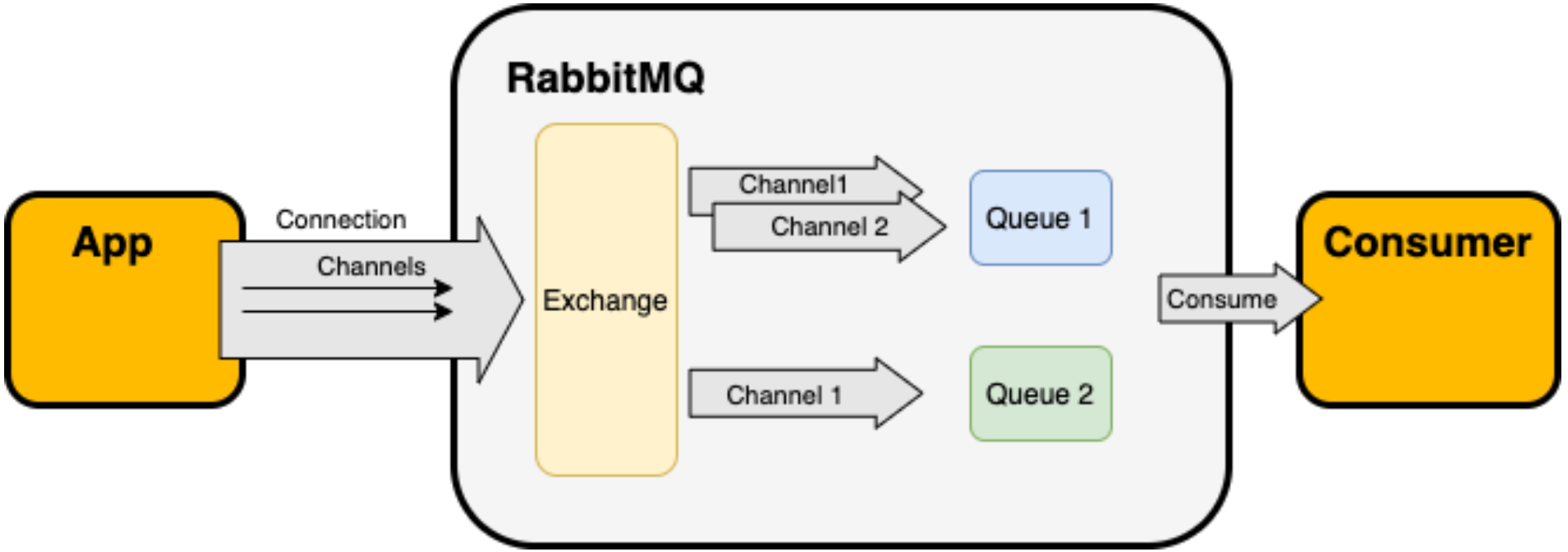
ДП.045440-07-99
Асинхронна черга завдань для бібліотеки AsyncIO Python.
Компоненти бібліотеки. UML-діаграма компонентів

Схема взаємодії бібліотеки Tabasco та RabbitMQ



Рябоконь Т.О., група КП-62

Схема роботи RabbitMQ



Рябоконь Т.О., група КП-62

Додаток 2

Лістинг головного модулю, модулю виконавця і планувальника завдань бібліотеки Tabasco

tabasco.py

```
import asyncio
import importlib
import json
import typing as t
import uuid

import aio_pika
from mypy_extensions import TypedDict

from tabasco.config import TabascoConfig
from tabasco.task import TabascoTask
from tabasco.task import TabascoTaskSignature


class TabascoPublishData(TypedDict):
    task_name: str
    task_id: str
    args: t.Sequence[t.Any]
    kwargs: t.Dict[str, t.Any]


class Tabasco:

    def __init__(
        self,
        name: str,
        config: TabascoConfig,
        *,
        loop: t.Optional[asyncio.AbstractEventLoop] = None,
    ) -> None:
        self.config = config

        self.name = name
        self.amqp_url = config['amqp_url']

        self.queue_name_prefix = config.get(
            'queue_name_prefix',
            f'{name}_tabasco',
        )
        self.default_queue_name = config.get(
            'default_queue_name',
            'default',
        )

        self.loop = loop if loop is not None else asyncio.get_event_loop()

        self._connection: t.Optional[aio_pika.Connection] = None
        self._channel: t.Optional[aio_pika.Channel] = None

        self.tasks: t.Dict[str, TabascoTask] = {}

    async def connect(self) -> None:
        if self._connection is not None:
            raise ValueError('Tabasco is already connected')

        self._connection = await aio_pika.connect_robust(
            self.amqp_url,
            loop=self.loop,
        )
        self._channel = await self._connection.channel()

    async def disconnect(self) -> None:
```

```
        if self._connection is None:
            raise ValueError('Tabasco is not connected')

        await t.cast(aio_pika.Channel, self._channel).close()
        await self._connection.close()

        self._channel = None
        self._connection = None

    @property
    def connection(self) -> aio_pika.Connection:
        if self._connection is None:
            raise ValueError('Tabasco is not connected')

        return self._connection

    @property
    def channel(self) -> aio_pika.Channel:
        if self._connection is None:
            raise ValueError('Tabasco is not connected')

        return self._channel

    def default_amqp_queue_name(self) -> str:
        return f'{self.queue_name_prefix}_{self.default_queue_name}'

    def task(
        self,
        func: t.Callable[..., t.Any],
    ) -> TabascoTask:
        task = TabascoTask(self, func)
        self.tasks[task.name] = task
        return task

    def discover(self, modules: t.Sequence[str]) -> None:
        for module in modules:
            importlib.import_module(module)

    async def publish(self, signature: TabascoTaskSignature) -> str:
        if signature.name not in self.tasks:
            raise RuntimeError(f'No such task: {signature.name}')

        task_id = str(uuid.uuid4())

        publish_data = TabascoPublishData(
            task_name=signature.name,
            task_id=task_id,
            args=signature.args,
            kwargs=signature.kwargs,
        )

        await self.channel.default_exchange.publish(
            aio_pika.Message(
                json.dumps(publish_data).encode(),
                delivery_mode=aio_pika.DeliveryMode.PERSISTENT,
            ),
            routing_key=self.default_amqp_queue_name(),
        )

        return task_id
```

worker.py

```
import asyncio
import json
import logging
import signal
import time
import typing as t

import aio_pika

from tabasco.tabasco import Tabasco


log = logging.getLogger(__name__)


class TabascoWorker:

    def __init__(
        self,
        app: Tabasco,
        *,
        concurrency: t.Optional[int] = None,
    ) -> None:
        self.app = app

        self.concurrency = concurrency if concurrency is not None else 10

        self.startup_hooks: t.Sequence[t.Callable[[], t.Awaitable[None]]] =
[]
        self.cleanup_hooks: t.Sequence[t.Callable[[], t.Awaitable[None]]] =
[]

        if (
            'worker' in self.app.config
            and 'startup_hooks' in self.app.config['worker']
        ):
            self.startup_hooks = self.app.config['worker']['startup_hooks']

        if (
            'worker' in self.app.config
            and 'cleanup_hooks' in self.app.config['worker']
        ):
            self.cleanup_hooks = self.app.config['worker']['cleanup_hooks']

        self._running_tasks = 0
        self._cleanup_cond = asyncio.Condition()

    async def _on_message(self, message: aio_pika.IncomingMessage) -> None:
        self._running_tasks += 1

        with message.process():
            publish_data = json.loads(message.body.decode())
            task = self.app.tasks[publish_data['task_name']]

            log.info(
                'Received task %s[%s]',
                publish_data['task_name'],
                publish_data['task_id'],
            )

            start_time = time.time()
            try:
```

```
                if task.is_async():
                    result = await task(
                        *publish_data['args'],
                        **publish_data['kwargs'],
                    )
                else:
                    result = await self.app.loop.run_in_executor(
                        None,
                        lambda: task(
                            *publish_data['args'],
                            **publish_data['kwargs'],
                        ),
                    )

                exec_seconds = time.time() - start_time

                log.info(
                    'Succeeded task %s[%s] in %.4fs: %s',
                    publish_data['task_name'],
                    publish_data['task_id'],
                    exec_seconds,
                    result,
                )
            except Exception:
                exec_seconds = time.time() - start_time

                log.exception(
                    'Task %s[%s] finished with error in %.4fs:',
                    publish_data['task_name'],
                    publish_data['task_id'],
                    exec_seconds,
                )

        async with self._cleanup_cond:
            self._running_tasks -= 1
            self._cleanup_cond.notify_all()

    async def _startup(self) -> None:
        await self.app.connect()

        for coro in self.startup_hooks:
            await coro()

        init_str = 'Starting worker. Registered tasks:\n'
        for task_name in self.app.tasks.keys():
            init_str += f'  - {task_name}\n'
        log.info(init_str)

        await self.app.channel.set_qos(prefetch_count=self.concurrency)

        self._queue = await self.app.channel.declare_queue(
            self.app.default_amqp_queue_name(),
            durable=True,
        )
        self._consumer_tag = await self._queue.consume(self._on_message)

    async def _cleanup(self) -> None:
        await self._queue.cancel(self._consumer_tag)

        async with self._cleanup_cond:
            while self._running_tasks > 0:
                log.info(
                    'Stopping worker. Waiting for %s tasks to stop',
                    self._running_tasks,
```

```
                )
                await self._cleanup_cond.wait()

        log.info('All tasks stopped. Stopping worker')

        for coro in self.cleanup_hooks:
            await coro()

        await self.app.disconnect()

        self.app.loop.stop()

    def _close(self) -> None:
        self.app.loop.remove_signal_handler(signal.SIGTERM)
        self.app.loop.remove_signal_handler(signal.SIGINT)
        self.app.loop.create_task(self._cleanup())

    def run(self) -> None:
        self.app.loop.add_signal_handler(signal.SIGTERM, self._close)
        self.app.loop.add_signal_handler(signal.SIGINT, self._close)

        self.app.loop.run_until_complete(self._startup())

        try:
            self.app.loop.run_forever()
        finally:
            self.app.loop.close()
```

beat.py

```
import asyncio
import logging
import math
import signal
import time
import typing as t
from datetime import timedelta
from datetime import timezone

from tabasco.config import TabascoBeatSchedule
from tabasco.crontab import TabascoCrontab
from tabasco.tabasco import Tabasco
from tabasco.task import TabascoTaskSignature


log = logging.getLogger(__name__)


class TabascoBeat:

    def __init__(self, app: Tabasco) -> None:
        if 'beat' not in app.config:
            raise RuntimeError(
                'You should provide "beat" config entry '
                'to run tabasco beat service'
            )
        if 'schedules' not in app.config['beat']:
            raise RuntimeError(
                'You should provide "schedule" config entry inside '
                '"beat" entry to run tabasco beat service'
            )

        self.app = app
```

```
            self.timezone = (
                self.app.config['beat']['timezone']
                if 'timezone' in self.app.config['beat']
                else timezone.utc
            )
            self.schedules = self.app.config['beat']['schedules']

            self._prev_time = math.floor(time.time())
            self._main_task: t.Optional[t.Awaitable[None]] = None
            self._should_stop = False

            self._period_schedules = [
                s
                for s in self.schedules
                if (
                    isinstance(s['schedule'], int)
                    or isinstance(s['schedule'], timedelta)
                )
            ]

            self._crontab_schedules: t.Dict[int, t.List[TabascoBeatSchedule]] =
{}
            for schedule in self.schedules:
                if isinstance(schedule['schedule'], TabascoCrontab):
                    self._put_crontab_schedule(schedule, self._prev_time)

        def _put_crontab_schedule(
            self,
            schedule: TabascoBeatSchedule,
            current_time: int,
        ) -> None:
            next_timestamp = (
                t.cast(TabascoCrontab, schedule['schedule'])
                .next_timestamp(current_time, self.timezone)
            )
            if next_timestamp not in self._crontab_schedules:
                self._crontab_schedules[next_timestamp] = [schedule]
            else:
                self._crontab_schedules[next_timestamp].append(schedule)

        async def _publish_task(self, schedule: TabascoBeatSchedule) -> None:
            if isinstance(schedule['task'], str):
                signature = TabascoTaskSignature(name=schedule['task'])
            else:
                signature = schedule['task']

            task_id = await self.app.publish(signature)

            log.info(
                'Task %s[%s] queued',
                schedule['task'],
                task_id,
            )

        async def _beat(self) -> None:
            current_time = math.floor(time.time())

            if current_time > self._prev_time:
                for schedule in self._period_schedules:
                    if isinstance(schedule['schedule'], timedelta):
                        period = int(schedule['schedule'].total_seconds())
                    else:
                        period = t.cast(int, schedule['schedule'])
```

```
            if current_time % period == 0:
                await self._publish_task(schedule)

    for ts in range(self._prev_time + 1, current_time + 1):
        if ts in self._crontab_schedules:
            for schedule in self._crontab_schedules[ts]:
                await self._publish_task(schedule)
                self._put_crontab_schedule(schedule, current_time)

            del self._crontab_schedules[ts]

    self._prev_time = current_time

async def _main(self) -> None:
    while True:
        asyncio.create_task(self._beat())

        await asyncio.sleep(1)

        if self._should_stop:
            break

async def _startup(self) -> None:
    await self.app.connect()

    init_str = 'Starting beat service. Registered tasks:\n'
    for task_name in self.app.tasks.keys():
        init_str += f'  - {task_name}\n'
    log.info(init_str)

    self._main_task = asyncio.create_task(self._main())

async def _cleanup(self) -> None:
    log.info('Stopping beat service')

    self._should_stop = True
    if self._main_task is not None:
        await self._main_task

    await self.app.disconnect()

    self.app.loop.stop()

def _close(self) -> None:
    self.app.loop.remove_signal_handler(signal.SIGTERM)
    self.app.loop.remove_signal_handler(signal.SIGINT)
    self.app.loop.create_task(self._cleanup())

def run(self) -> None:
    self.app.loop.add_signal_handler(signal.SIGTERM, self._close)
    self.app.loop.add_signal_handler(signal.SIGINT, self._close)

    self.app.loop.run_until_complete(self._startup())

    try:
        self.app.loop.run_forever()
    finally:
        self.app.loop.close()
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

АСИНХРОННА ЧЕРГА ЗАВДАНЬ ДЛЯ БІБЛІОТЕКИ ASYNCIO PYTHON

Виконала: Рябоконь Тетяна Олексіївна

Керівник: доц. каф. ПЗКС, доц., к.т.н. Заболотня Тетяна Миколаївна

Київ – 2020

АКТУАЛЬНІСТЬ

Поява **asyncio Python** та групи бібліотек **aio-libs**

Типові задачі черги завдань:

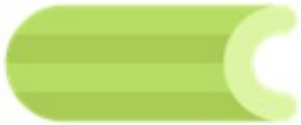
періодичні завдання, завдання, що не потребують участі користувача

Потреба в бібліотеці, що відповідає всім вимогам до черги завдань і працює з **asyncio**

Опис існуючих рішень

- Виконання завдань в різних чергах
- Виконання за розкладом
- Керування за допомогою CLI

- Працює на базі asyncio
- Має покриття типами
- Має просту реалізацію
- Використовує uvloop



Постановка задачі

Мета проекту: створення асинхронної черги завдань для асинхронного Python, що буде працювати на базі модулю стандартної бібліотеки asyncio

Асинхронна черга завдань:
Реалізація основної функціональності, що має надавати інструмент такого типу

Швидкість:
Швидкість програм на asyncio можна ще пришвидшити за допомогою uvloop

Tabasco

Статична типізація:
Бібліотека має бути покрита анотаціями типів та перевірена за допомогою mypy

asyncio

Асинхронність:
Бібліотека має працювати на основі asyncio та використовувати синтаксис async/await

Засоби розроблення

- бібліотека asyncio;
- бібліотека aio_pika;
- цикл подій uvloop;
- інструмент перевірки типів myру.

asyncio

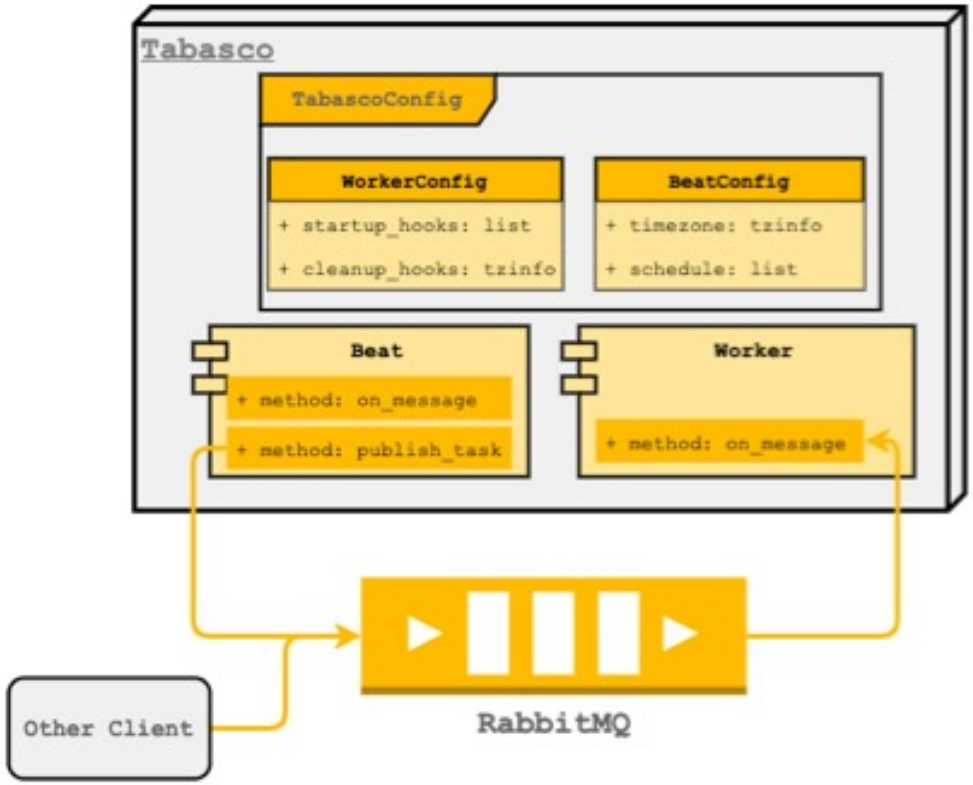
: my[py]

- асинхронний Python
- можливість взаємодіяти з RabbitMQ;
- можливість використання циклу подій uvloop замість стандартного в asyncio;
- перевірка типів.

- збільшення швидкості додатку;
- використання RabbitMQ в якості брокера повідомлень;
- збільшення читаємості коду та додаткова перевірка коду.

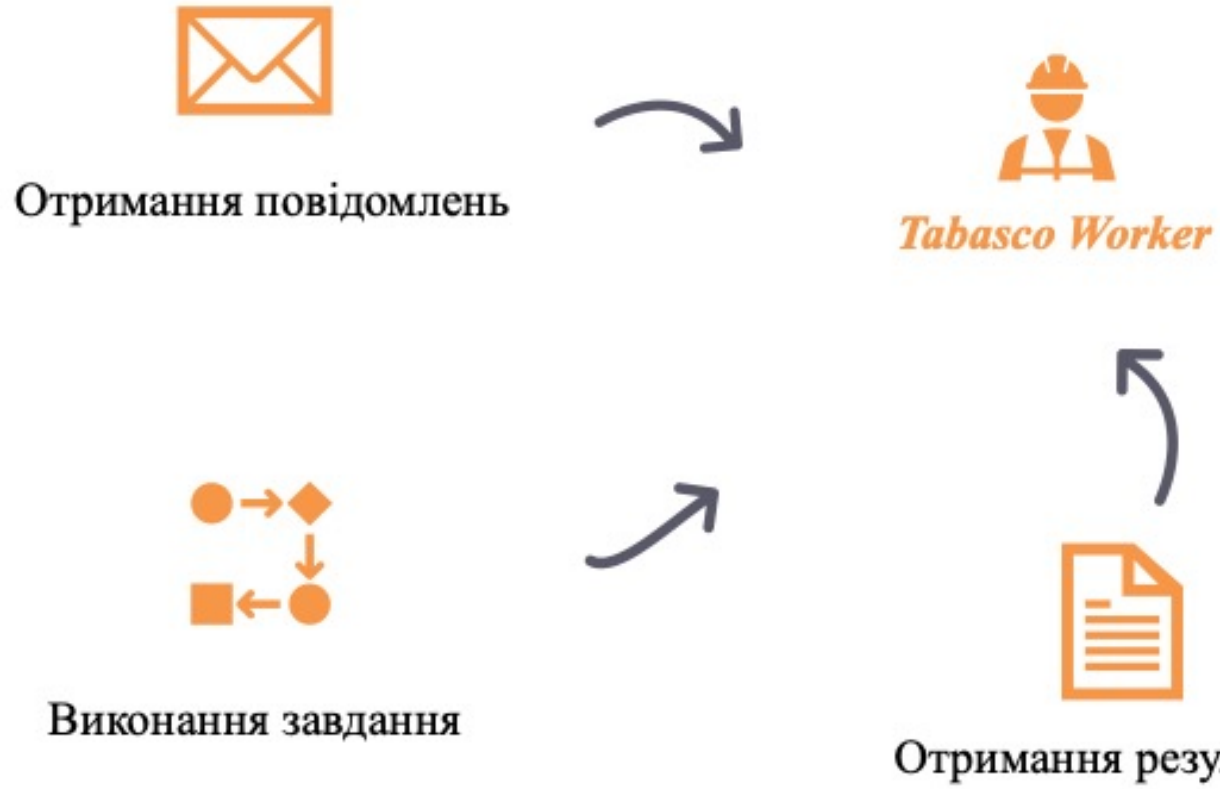
aio-pika

Опис розроблених програмних засобів



- Загальна схема взаємодії Tabasco та RabbitMQ
- Схема роботи RabbitMQ

Функції виконавця завдань



Алгоритм роботи виконавця завдань

Функції планувальника завдань

Визначення графіку виконання періодичних завдань

Tabasco Beat

Обчислення часу наступного виконання завдання

Надсилання завдань виконавцю

Алгоритм роботи планувальника завдань

Приклади використання бібліотеки

Приклад створення завдання

```
from datetime import datetime
from habanero.tabasco import app as tabasco_app

@tabasco_app.task
async def answer_to_ultimate_question_of_life(answer: str = '42') -> str:
    return f'Answer to the Ultimate Question of Life is {answer}'
```

Приклад надсилання завдання в чергу

```
from habanero.tabasco import app as tabasco_app
from habanero.tasks.examples_tasks import answer_to_ultimate_question_of_life

async def main() -> None:
    # Підключення до RabbitMQ
    await tabasco_app.connect()

    # Відправлення завдання до черги:
    result = await answer_to_ultimate_question_of_life.delay(answer='Tabasco!')

    # Розрив з'єдання з RabbitMQ
    await tabasco_app.disconnect()
```

Приклад налаштування додатку бібліотеки Tabasco

```
# секція імпортів

tabasco_config = TabascoConfig(
    amqp_url=config['amqp']['url'],
    worker=TabascoWorkerConfig(startup_hooks=[], cleanup_hooks=[]),
    beat=TabascoBeatConfig(
        timezone=timezone.tz,
        schedules=[
            TabascoBeatSchedule(
                task=(
                    'habanero.tasks.examples_tasks.'
                     'list_scoville_table_lines'
                ),
                schedule=TabascoCrontab(second=2),
            ),
        ],
    )
)

app = Tabasco('habanero', tabasco_config, loop=loop)
```

Приклади використання CLI

Запуск виконавця та планувальника завдань відповідно:

```
python -m tabasco -A habanero.tabasco.app worker -l info error
python -m tabasco -A habanero.tabasco.app beat -l info error
```

де -A – екземпляр додатку асинхронної черги завдань,
-l – рівень логування (error, info та ін.)

Команда виводу всіх зареєстрованих в додатку завдань:

```
python -m tabasco -A habanero.tabasco.app tasks
```

Команда відправки завдання в чергу:

```
python -m tabasco -A habanero.tabasco.app delay
habanero.tasks.examples_tasks.list_scoville_table_lines
```

Апробація

МІЖНАРОДНИЙ МОЛОДІЖНИЙ ФОРУМ
«РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ У XXI СТОЛІТТІ»
КОНФЕРЕНЦІЯ «ІНФОРМАЦІЙНІ ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ»
«Програмна інженерія. Інформаційні технології в освіті»

NURE
Харківський національний університет радіоелектроніки

Підготовлено пакет документів на отримання авторського свідоцтва

14/18

ВИСНОВКИ

Розглянуто проблему, поставлено задачу і досягнуто ціль розроблення асинхронної черги завдань для модулю стандартної бібліотеки asyncio Python, що покрита анотаціями типів, перевірена за допомогою mypy та використовує синтаксис async/await.

Розроблений програмний засіб забезпечує:

- виконання тривалих завдань в окремому процесі;
- виконання завдань за графіком;
- роботу з asyncio Python.

Напрямки подальшої роботи:

- більш гнучка реалізація парсеру налаштування графіку виконання періодичних завдань;
- можливість налаштування багатьох черг виконання завдань.

Дякую за увагу!

Схема взаємодії Tabasco та RabbitMQ

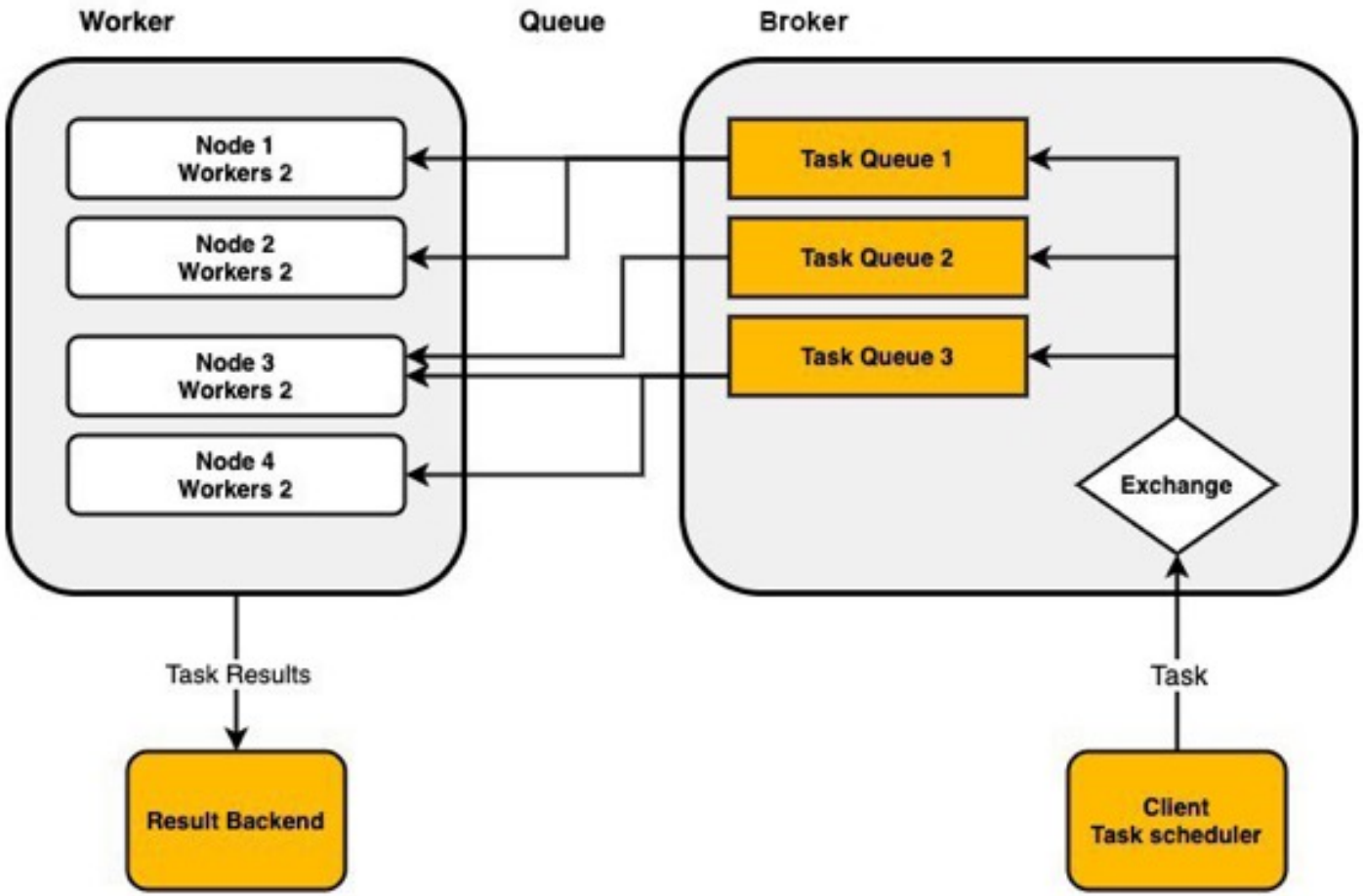
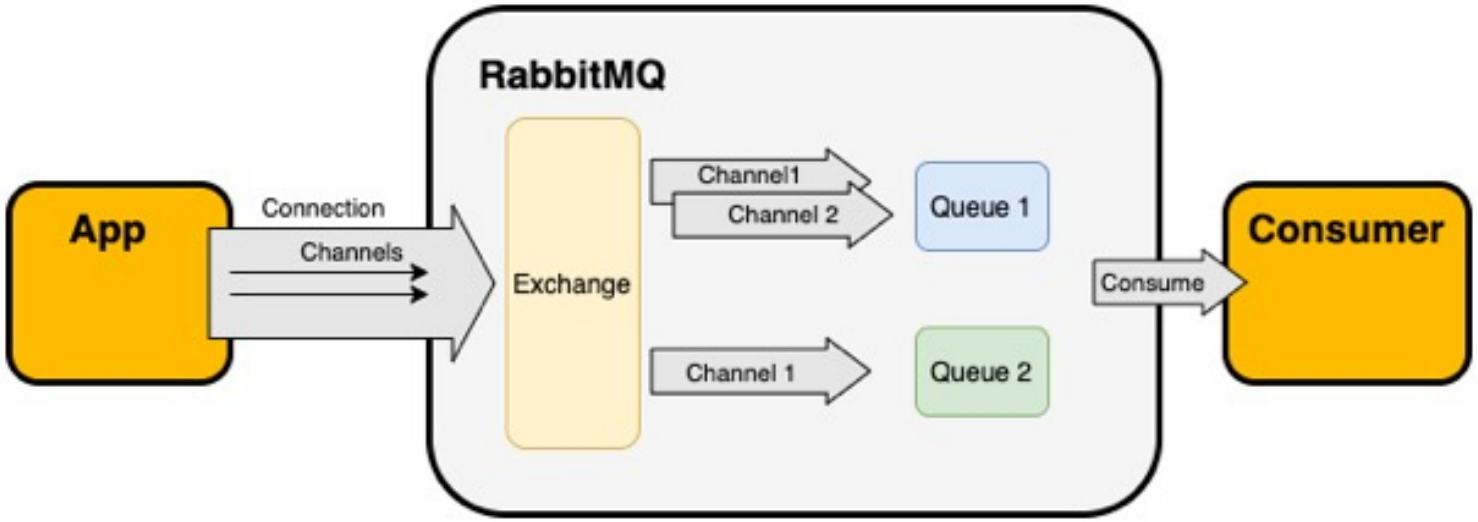


Схема роботи RabbitMQ



Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»

Науковий керівник кафедри

_____ Іван ДИЧКА

«___» _____ 2019 р.

АСИНХРОННА ЧЕРГА ЗАВДАНЬ ДЛЯ БІБЛІОТЕКИ ASYNCIO PYTHON

Програма та методика тестування

ДП.045440-04-51

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Тетяна РЯБОКОНЬ

2019

ЗМІСТ

1. Об’єкт випробувань……...3
2. Мета тестування…….3
3. Методи тестування…….…3
4. Засоби та порядок тестування……...4

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Бібліотека асинхронна черга завдань, що працює з модулем стандартної бібліотеки Python – AsyncIO.

2. МЕТА ТЕСТУВАННЯ

У процесі тестування має бути перевірено наступне:

1) працездатність та коректна робота виконавця завдань;
2) працездатність та коректна робота планувальника завдань;
3) коректна робота інтерфейсу командного рядку;
4) забезпечення належного рівня відмовостійкості та масштабування черги завдань;
5) відповідність розробки вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування відбувається за допомогою методу Grey Box Testing. Grey Box Testing - це методика тестування програмного продукту з частковим знанням внутрішньої роботи програми, метою такого тестування є пошук дефектів через помилки в логіці коду або неправильного використання програми.

Для тестування даної бібліотеки використовуються такі методи:

1) функціональне тестування на рівні Smoke Testing;
2) тестування продуктивності програмного забезпечення методами Failover and Recovery Testing (тестування на відмову та відновлення) та Load testing (навантажувальне тестування);
3) покриття коду.

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Працездатність черги завдань перевіряється шляхом:

1) динамічного ручного тестування на відповідність функціональним вимогам;
2) статичного тестування коду;
3) тестування бібліотеки в різних операційних системах;
4) тестування стабільності роботи при граничних умовах;
5) тестування при великому навантаженні;
6) тестування інтерфейсу командного рядку;
7) тестування масштабованості бібліотеки.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

АСИНХРОННА ЧЕРГА ЗАВДАНЬ ДЛЯ БІБЛІОТЕКИ ASYNCIO PYTHON

Керівництво програміста

ДП.045440-05-33

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Тетяна РЯБОКОНЬ

2020

ЗМІСТ

1. Призначення і умови застосування програми..... 3
2. Характеристики програми 6
3. Звернення до програми 7
4. Вхідні та вихідні дані 10
5. Повідомлення 11

1. Призначення і умови застосування програми

1.1. Призначення програми

Дана бібліотека імплементує розподілену асинхронну чергу завдань. Черги завдань керують фоновою роботою, яка повинна виконуватися поза звичайним циклом запитів-відповідей HTTP.

Черги завдань модуль бути корисними, коли потрібно виконати деяку роботу яка не ініційована HTTP-запитом, або деяке тривале завдання, що різко знизить продуктивність та швидкість HTTP-відповіді додатку.

Наприклад, веб-додаток має опитувати деяке стороннє API кожні 10 хвилин, щоб зібрати деяку інформацію необхідну йому для роботи. Черга завдань може виконувати код, що буде посилати запити на стороннє API, та зберігати необхідні дані до бази даних. Таким чином веб-додаток буде розвантажений і час відклику значно зменшиться. Черги завдань створені, щоб виконувати велику кількість фонових завдань в додатках та для виконання завдань за графіком.

Бібліотека розроблена в рамках даного дипломного проєкту відповідає основним вимогам, що постають перед чергою завдань. Її основна відмінність в тому, що вона працює з асинхронним Python, а саме з модулем стандартної бібліотеки AsyncIO.

Інші типи завдань для черг завдань включають:

- розподілити в часі велику кількість записів в базу даних замість того, щоб виконувати всі відразу;
- оброблення деяких даних за фіксований інтервал часу, наприклад, кожні 15 хвилин;
- планування періодичних завдань, наприклад, оновлення деяких даних в базі даних.

1.2. Функції, що виконує програма

Основою функцією програми є виконання в реальному часі деяких завдань необхідних для роботи програмного додатку, але в окремому процесі, або навіть на окремому сервері. Також програма потрібна для того, щоб додаток міг витримувати великі навантаження, за допомогою збільшення кількості виконавців завдань.

Також важливою функцією даної бібліотеки є виконання завдань за певним графіком. Виконання завдань можна налаштувати з точністю до секунди.

Отже, основними функціями даної бібліотеки є:

- виконання тривалих фонових завдань у реальному часі, але в окремому процесі;
- виконання завдань за графіком;
- масштабування для витримування більшого навантаження;
- забезпечення надійності даних, завдяки автовідновлюваному з'єднанню та процедурі коректного завершення роботи всіх компонентів програми.

1.3. Умови необхідні для виконання програми

Для роботи даної бібліотеки потрібні деякі сторонні залежності:

- aio-pika – обгортка над асинхронним клієнтом для AMQP 0.9.1 – aiormq, що працює з AsyncIO;
- idna - підтримка інтернаціоналізованого протоколу доменних імен в додатках (IDNA), визначеного в RFC 5891. Це остання версія протоколу, яку іноді називають "IDNA 2008".
- multidict – пакет, що імплементує колекцію ключ-значення, схожу на словник в Python, де ключ може зустрічатися в контейнері більше ніж один раз;
- shortuuid – проста Python бібліотека, яка генерує стислі, однозначні, безпечні для URL UUID;

- yarl – проста Python бібліотека для роботи з URL.

Код бібліотеки був перевірений за допомогою лінтеру flake8 та інструменту перевірки типів mypy. Для них також потрібні деякі залежності:

- mypy – інструмент для перевірки статичних типів в Python;
- flake8 – інструмент для перевірки коду на відповідність конвенцій Python (PEP8 та ін.);
- flake8-import-order – розширення для flake8, що перевіряє порядок імпортів в коді;
- mypy-extensions – модуль, що визначає експериментальні розширення до стандартного модуля typing, які підтримуються інструментом перевірки типу mypy.;
- typed-ast – це пакет Python 3, який забезпечує аналізатор Python 2.7 та Python 3, подібний до стандартної бібліотеки ast. На відміну від ast, парсери в typed_ast включають коментарі типу PEP 484 і не залежать від версії Python, під якою вони запускаються.;
- pycodestyle – це інструмент для перевірки Python-коду на відповідність деяким стильовим умовам у PEP 8.;
- pyflakes – бібліотека, яка перевіряє вихідні файли Python на наявність помилок.;
- entrypoints – це спосіб для пакетів Python визначати об'єкти з деяким загальним інтерфейсом. Найпоширеніші приклади – точки входу console_scripts, які визначають команди оболонки, визначаючи функцію Python, яку потрібно запустити.

Окрім цього, бібліотека працює на основі брокеру повідомлень RabbitMQ, тому він повинен бути розгорнутий для роботи бібліотеки.

Даний програмний продукт є контейнеризованим за допомогою Docker, а для роботи на локальній машині використаний інструмент docker-compose, тому всі необхідні залежності будуть встановлені автоматично.

2. Характеристики програми

2.1. Опис основних характеристик програми

2.1.1. Режим роботи програми

Дана бібліотека складається з двох основних частин: планувальника та виконавця завдань.

Виконавець завдань працює як однопотоковий демон. Демон – це комп'ютерна програма, яка працює як фоновий процес, і не знаходиться під безпосереднім контролем користувача. Можна запустити більше одного виконавця для горизонтального масштабування. Виконавець приймає завдання від клієнтів або від планувальника, виконує їх та повертає їм результат.

Планувальник завдань також працює як демон, він раз на секунду перевіряє чи співпадає поточний час с заданими налаштуваннями графіків зареєстрованих в додатку завдань, якщо він знаходить співпадіння, то відправляє заплановане на даний час завдання виконавцеві.

2.1.2. Контроль правильності виконання програми

Працездатність розробленої бібліотеки можна перевірити наступним чином:

- запустити виконавця і надіслати завдання за допомогою CLI команди delay;
- запустити виконавця і надіслати йому завдання за допомогою програмного додатку клієнта;
- запустити виконавця та планувальника, дочекатися доки настане час виконання хоча б одної з зареєстрованих в додатку завдань.

Коли виконавець отримає завдання, він сповістить про це вивівши повідомлення про початок виконання, а коли він виконає завдання, то сповістить про результат або помилку.

2.2. Опис основних особливостей програми

Дана бібліотека має інтерфейс командного рядку Для запуску додатоку асинхронної черги завдань треба скористатися командою tabasco та вказати шлях до об'єкту екземпляру додатку черги в основному додатку, з яким працює черга. Також доступні підкоманди worker, beat, tasks та delay.

- команда worker запускає екземпляр виконавця та передає йому караметри конкурентності та рівня логування;
- команда beat запускає планувальник завдань та приймає на вхід рівень логування;
- команда delay надсилає задання виконавцеві і приймає такі параметри: ім’я завдання, позиційні аргументи та аргументи за ключовими словами;
- за допомогою команди tasks можна переглянути всі зареєстровані в додатку завдання.

Кожна команда має справку, що робить CLI більш зрозумілим для користувача.

3. Звернення до програми

3.1. Запуск програми

Бібліотека може бути запущена за допомогою CLI команди tabasco.

Приклад запуску бібліотеки

```
python -m tabasco -A habanero.tabasco.app worker -l info error
```

Також для забезпечення консистентної роботи бібліотеки на різних платформах, її було контейнеризовано за допомогою Docker. Для зручності розроблення бібліотеки був використаний docker-compose, тому виконавець та планувальник завдань можна запустити відповідними командами.

Команда запуску виконавця завдань

```
docker-compose up tabasco-worker
```

Команда запуску планувальника завдань

```
docker-compose up tabasco-beat
```

Також бібліотека покрита типами і перевірена за допомогою mypy, а стиль коду перевіряється за допомогою flake8. Запустити ці інструменти можна за допомогою відповідних команд

Команди запуску flake8 та mypy відповідно

```
docker-compose up flake8
docker-compose up mypy
```

3.2. Виконання програми

3.2.1. Робота виконавця завдань

Виконавець завдань (worker) потрібен для того, щоб забирати завдання з черги для подальшого їх виконання. Модуль виконавця відповідає за отримання повідомлень, виконування завдань, відправку результатів, обробку сигналів, налаштування логування тощо.

Цей модуль має такі методи:

- _on_message – метод, що приймає повідомлення, яке надійшло в канал, з яким працює виконавець, знаходить у повідомленні ім’я завдання та по знайденому імені викликає Python функцію, що зареєстрована в додатку черги під відповідним ім’ям;
- _startup – виконується при запуску виконавця, встановлює з'єднання з RabbitMQ та викликає всі методи, що задані в конфігураціях і повинні бути виконані на старті та назначає функцію _on_message обробником надходження повідомлення в чергу;
- _cleanup – виконується перед завершенням роботи виконавця.

3.2.2. Робота планувальника завдань

Модуль планувальника передивляється всі завдання та вибирає ті, що виконувалися останній раз рівно стільки часу назад, скільки вказано в конфігурації. Після відправки завдання на виконання, планувальник встановлює час його наступного виконання.

- _put_crontab_schedule – метод, який встановлює час наступного виконання завдання;
- _publish_task – збирає сигнатуру завдання та публікує його в чергу брокера, з якою працюють планувальник та виконавець;
- _beat – обирає завдання, яке треба виконати згідно з графіком і за допомогою виконання методу _put_crontab_schedule планує наступне виконання даного завдання;
- _main – викликає метод _beat кожну секунду;
- _startup – виконується при запуску планувальника, встановлює з'єднання з RabbitMQ та викликає метод _main.

3.2.3. Налаштування графіку виконання періодичних завдань

Модуль crontab дозволяє налаштовувати графік виконання періодичних завдань. Модуль має метод – next_timestamp, який приймає поточний час та повертає час наступного виконання завдання. Даний модуль приймає конфігурації графіку виконання для кожного завдання, що зареєстроване в ньому. Ці налаштування можуть бути задані за допомогою відповідних параметрів, таких як: day, hour, minute та second.

3.3. Завершення роботи програми

В даній бібліотеці передбачене коректне завершення роботи для виконавця і планувальника завдань. Воно реалізоване за допомогою таких методів:

- _cleanup – виконується перед завершенням роботи, розриває з’єднання з RabbitMQ, зупиняє цикл подій;

- _close – прибирає обробники сигналів SIGTERM та SIGINT, що були встановлені на початку роботи, викликає метод _cleanup.

Виконавець завдань при завершенні своєї роботи очікує на завершення виконання всіх завдань, що були взяті ним до роботи.

Також перед завершенням проводиться закриття з’єднання з RabbitMQ.

4. Вхідні та вихідні дані

4.1. Організація вхідної інформації, що використовується програмою

Виконавець завдань приймає на вхід конфігурацію, що містить функції, які потрібно виконати при старті та при завершенні його роботи. Також він отримує значення конкурентності, тобто значення, що встановить “prefetch count” для RabbitMQ. Це потрібно для забезпечення можливості обмежувати кількість непідтверджених повідомлень на каналі при обробці повідомлення. Значення за замовчуванням для цього параметру дорівнює 10.

Планувальник завдань приймає на вхід конфігурацію, яка містить налаштування графіків виконання завдань та часовий пояс. Конфігурація обов’язково повинна містити розклади завдань, інакше модуль не зможе коректно працювати.

4.2. Організація вихідної інформації, що використовується програмою

Частини даної бібліотеки передають, приймають, виконують та повертають результат завдання. В чистому вигляді жодна з її частин не повертає ніяку інформацію.

5. Повідомлення

Виконавець завдань надсилає наступні текстові повідомлення:

- *Starting Worker. Registered tasks: <список всіх зареєстрованих завдань>* – повідомлення на початку роботи виконавця завдань;
- *Received task <ім'я завдання>[<ідентифікатор завдання>]* – при отриманні завдання;
- *Succeeded task <ім'я завдання> [<ідентифікатор завдання>] <час виконання завдання> <результат виконання>* – при успішному виконанні завдання;
- *Task < ім'я завдання> [<ідентифікатор завдання>] finished with error in <час виконання завдання>* – при завершенні завдання з помилкою.
- *Stopping worker. Waiting for <кількість незавершених завдань> tasks to stop* – при завершенні роботи виконавця, коли ще є незавершенні взяті до роботи завдання;
- *All tasks stoppped. Stopping worker* – при остаточному завершенні роботи виконавця завдань.

Планувальник завдань надсилає повідомлення тільки при старті та завершенні своєї роботи. Повідомлення аналогічні повідомленням виконавця завдань.